修士論文

許可制ブロックチェーンと既存分散データベースとの性能・実装・運用比較

早稲田大学大学院基幹理工学研究科
情報理工・情報通信専攻

池内弘樹

学籍番号　5115F006-4

提出年月日　2017 年 1 月 30 日

指導教授　中島達夫

Comparing the permissioned blockchain and the existing distributed database in term of peformance, implementation and operation

Kohki Ikeuchi

Thesis submitted in partial fulfillment of

the requirements for the degree of

Master in Computer Science

Student ID	5115F006-4
Submission Date	Jan 30. 2017
Supervisor	Prof. Tatsuo NAKAJIMA

Department of Computer Science School of Science and Engineering WASEDA University

概要

現在ブロックチェーンが新しい通貨形態としてだけでなく，ネットワークトラフィックを削減し，中央集権型システムの冗長化と可用性のための汎用的な分散化を行うプラットフォームとしても注目されている。例えば IoT 化と組み合わせて使うことなどが提案されている。汎用的に利用しようとすると分散データベースの一面が大きくなるが，既存の分散データベースとはデータの分散の仕方やデータ構造，コンセンサスのとり方が大きく異なり使い分けが必要となる。そこで本研究では汎用的な利用が考えられている許可制ブロックチェーンである Hyperledger Project の Fabric と，既存の分散データベースである Cassandra とをそれぞれ用いアプリケーションを作成する。この 2 つのアプリケーションは同等の機能を持つ。その後性能，実装，運用の面で比較を行う。

結果として Fabric を用いたアプリケーションではブロックチェーンのデータモデルに起因してトランザクション処理が増えることがわかった。Cassandra を用いたアプリケーションと比べると一定のレスポンスの速さはあるが，ネットワーク全体の取引量が増えると資産のリアルタイム性がなくなり可用性が下がることがわかった。また Fabric を用いると実装は少なく済むが，運用においてはその特性から保守性が下がることがわかった。このことから許可制のブロックチェーンは，取引を行うシステムを信頼性と品質を保ち容易な開発を可能とするプラットフォームとして重要となると考えられる。

Abstract

Currently, blockchain is attracting attention not only as a new currency type, but also as a platform for reducing network traffic and making generic decentralization for redundancy and availability of centralized systems. For example, it is proposed to use it in combination with IoT. If you try to use generically blockchain, one side of the distributed database becomes similar, however, how to distribute data, data structure and consensus are greatly different from the existing distributed database, and it is necessary to use properly. Therefore, in this research, we use Fabric of Hyperledger Project which is the permissioned blockchain considered to be general purpose, and Cassandra which is an existing distributed database, to create applications respectively. These two applications have equivalent functions. Then I compare these two applications in terms of performance, implementation and operation.

As a result, transaction processing is increased due to blockchain data model in application using Fabric. Although it has a certain response speed compared with the application using Cassandra, I found that the real-time property of the asset gets lost and availability drops as the transaction requests of the entire network increases. Also, it reduces implementation to use Fabric, however, it turned out that maintainability was reduced in operation. Therefore, it is thought that the permissioned blockchain is important as a platform that makes it possible to develop a system that conducts transactions with reliability and quality and easy development.

目次

図目次

表目次

1 序論

1.1 背景

現在ネットワークに接続されているコンピュータはコンピュータの小型化によるスマートフォンの浸透やコンピュータの安価化により様々な場面において利用されることが増えたことにより，その数は膨大になっている。その為ネットワーク全体のトラフィックは現状においても膨大な量となっている。その数は今後より小型化，安価化が進みまたコンピュータのクラウド化により Internet of Things(IoT)モノのネットワーク化の普及も相まって増加の一途を辿る。クラウド化が進むことによりリソースの最適化は進むとされるが[1]，ネットワーク全体のトラフィックは増加するとされており[2]，現状のネットワークにおいて追いつかなくなることが予測される[3]。そこで既存の通信において無駄となっており省略できる通信がないか見直しがされている。その中の一つに第三者機関による認証を必要とし多くのやり取りを必要とされている金融取引がある。既存の取引では多くの認証や合意形成を必要としているため，多くの通信を必要としている。そこで分散台帳技術であり分散型ネットワークであるブロックチェーンがビットコインを期に注目されている。

ブロックチェーンの技術は，Bitcoin のような新しい通貨体系として脚光を浴びたが，最近では Linux Foundation がホストとなりブロックチェーン技術の推進を進め，様々な分野に適用していこうとされている。その中には IoT の有用的な利用も含まれている[4]。Linux Foundation は Hyperledger Project の Open Blockchain[5]により，汎用的に利用できる基盤としてのブロックチェーンシステムの提供を行うことを目指している。これにより中央集権のネットワークモデルでは問題となっているシステムの置き換えが可能となり，可用性，信頼性の高いシステムにできると主張されている。Bitcoin[6]では匿名性の強いネットワーク上での利用つまり不特定多数が参加するネットワークモデルが想定されている為，コンセンサスモデルに Proof Of Work という仕事量によって参加者の正当性を確認する方法が取られている。その為処理に時間のかかることや参加者の数が多くなければネットワークの信頼性が担保されないなど限定的な条件を抱えていた。一方 Open Blockchain では参加ピアは事前に決まっていることが想定されているため，コンセンサスモデルに Practical Byzantine Fault Tolerance アルゴリズムが採用されまたコンセンサスモデルを選択可能であり，汎用性が高くなっている。このようにブロックチェーン技術を，Bitcoin で問題となっている部分を限定的にすることで対処することにより，その条件下において汎用的なものにし，通貨のような特化した大規模なシステムだけでなく，一般的にアプリケーションを作成するミドルウェア技術の選定の一つとしてブロックチェーン技術へと昇華しようとしている。これにより，ブロックチェーンの利用場面が増加していくことが考えられる。

1.2 目的

ブロックチェーンには各ピアが一貫性のあるデータを所持するため分散データベースの一面を持つ。今研究の目的は汎用性の高いブロックチェーンと既存の分散データベースとの違いを性能，実装，運用の面から明らかにすることである。それにより既存の分散データベースとブロックチェーンの適した選択場面の違いを明確にする。

ブロックチェーンは取引を全てブロックと呼ばれる取引をまとめたデータ構造上に記録していき，このブロックがチェーンのように連ねていく。その為一つ一つの取引をやり取りするのではなく，今までの取引全てをやり取りし追加していくことになり，これにより過去の取引の正当性がデータ自体に含まれることになる。また多くの分散データベースと違い全てのデータを全てのピアでもつ。このように分散データベースとはデータの分散方法や通信方法と内容が異なる。一方で Open Blockchain では汎用性が売りとされており，様々な場面での利用を想定している。そこで汎用的であるブロックチェーンを分散データベースとして見た際の性能を調べ，違いを明らかにする。この際ブロックチェーンにはその特性であるアセットの管理の信頼性を担保する仕組みや認証する仕組みが含まれるため，分散データベースにも同様の仕組みを確保することにより，他のミドルウェアとの比較を行っていく。またミドルウェアとして重要である開発と運用の容易さの調査も同時に行う。

2 調査

2.1 概要

ブロックチェーンとデータベースが同じ構成で利用する際に，どのような性能や実装，運用の違いが得られるか調査する。その為にまず現存するブロックチェーン技術と既存の分散データベースの調査をする。ブロックチェーンと一重に言っても様々な種類があり，また既存の分散データベースにおいても様々な種類がある。ブロックチェーンとはどのようなものか，どのような種類があるのか説明し，分散データベースについても同様に説明する。そしてその中からどの技術にするか選定する。

2.2 ブロックチェーン

ブロックチェーン技術とはネットワークで接続されたコンピュータ(ピア)で構成される分散処理，分散データ管理を行う為の技術であり，変更不可能な形でデータを保持する分散データベースである。ただし分散データベースではデータを分散して持つのが，基本的にブロックチェーンでは重複してデータを持つ。ブロックチェーンの技術は主に 4 つから構成されている。

- **台帳分散**

  ピア間で共有されるデータであり，取引がチェーンのように保存されていく台帳。分散/相互検証よって内容を保証する。取引記録と検証が行われる技術。

  分散/相互検証よって内容を保証する。取引記録と検証が行われる技術。

- **スマートコントラクト**

  ブロックチェーン上で動くプログラムであり決定的なロジック。契約条項が組み込まれ取引の規定，実行，処理がされる機能。

- **合意形成**

  ピアネットワークでトランザクションの合意と，トランザクションの正当性を確認が行われる技術。これによりピアネットワークで正しい一貫性が取られた状態を確保する。

- **暗号技術**

  電子署名機能や認証機能になどにより取引の正当性と機密性を確保する技術。

4 つの技術の中で特に注目されブロックチェーンらしさとなっているのは台帳分散とスマートコントラクトである。台帳分散のような方法でのデータの分散方法は既存の分散データベースとは大きく異なっている。またスマートコントラクトは取引を行うことに特化した機能であり，元来の分散データベースには備え着いていない機能である。一方で合意形成は分散データベースでも重要な要素であり，様々な方法が考えられている。

ブロックチェーンはネットワーク形態から大きく以下の 3 つに分類することができる。

- **パブリック型**（Bitcoin, Enigma, Ethereum）
- **コンソーシアム型，プライベート型**（Hyperledger, Ripple）

パブリック型は参加者が不特定多数であり，ネットワークを形成するピアは互いを知っている必要がない。つまり悪意のある参加者もネットワークにいることが想定される。パブリック型は非許可制ブロックチェーンとも呼ばれる。一方コンソーシアム型，プライベート型では許可された特定の参加者のみが想定されているため，信頼が担保されたネットワーク形成になる。 許可制ブロックチェーンとも呼ばれる。この様にネットワークの形成する方法が異なり使用される前提が全く異なる。その為参加者を疑う必要があり信頼性を確認しないといけないか，それとも参加者は信頼が確証されており必要が無いか違いがあり，合意形成の方法(コンセンサスモデル)も異なってくる。

パブリック型では Bitcoin(Bitcoin: A Peer-to-Peer Electronic Cash System[6])を始め，Proof Of Work により合意を取る。これは多数決の際のピアの重み付を行ってからコンセンサスをとる仕組みとなっている。ピアの重み付けの方法には様々あり，Proof of Stake(Proof of Activity: Extending Bitcoin’s Proof of Work via Proof of Stake[7])などが発案されている。この方法ではデータの一貫

性を取ろうとコンセンサスを取る際にまずネットワークを形成するピアの重み付けの評価が走る必要があるために，基本的に取引を行うトランザクション処理に時間がかかる。また，不特定多数で構成されるネットワークの為，コンセンサスモデルからピアの数がネットワークの信頼に繋がるが，その分ネットワーク全体で起きる処理が増える為問題になっている。

コンソーシアム型，プライベート型では事前に決定された参加者から形成されるネットワークであるという性質を利用し，Practical Byzantine Fault Tolerance[8]によってコンセンサスをとる仕組みとなっている。これにより少ないピアで信頼性の担保を確保するのと同時に，ピアの重み付けの評価が必要でなくなる。これにより取引時にかかるトランザクション処理にかかる時間を短くしている。閉じたコミュニティでの利用を想定することにより，パブリック型のブロックチェーンよりも小規模な分散アプリケーションも可能となり，汎用的なミドルウェアを目指している。

また従来のパブリック型のブロックチェーンとコンソーシアム型，プライベート型では CAP の定理[9]のどれに重点を置くかが異なっている事が多い。CAP の定理とは一貫性(Consistency，どのマシンからも同じデータであること)，可用性(Availability，常にサービスが利用可能であること)，分断耐性(Partition-tolerance，ネットワーク障害が起きてもシステム全体が間違った結果をかえさないこと)のうち2つのみが実現可能であるというものである。従来のパブリック型のブロックチェーンではその特性上，可用性と分断耐性に重点が置かれていたが，コンソーシアム型，プライベート型のブロックチェーンでは一貫性と分断耐性に重点が置かれている事が多い。

今研究では汎用的な利用が期待され，従来の分散データベースを利用したアプリケーションと似たような利用が想定されるコンソーシアム型，プライベート型のブロックチェーンを利用する。その中で OSS のプロジェクトであり Linux Foundation が中心となり，30 社以上(Accenture, ANZ Bank, Cisco, CLS, Credits, Deutsche Borse, Digital Asset Holdings, DTCC，富士通，IC3, IBM, Intel, J.P.Morgan，ロンドン証券取引所グループ，三菱 UFJ フィナンシャルグループ (MUFG), R3, State Street, SWIFT, VMware, Wells Fargo など)が協力しブロックチェーン技術と P2P 分散レッジャー技術の確立を目指している Hyperledger Project[1][5]を選択する。

2.3 Hyperledger Project

Hyperledger Project には Sawtooth Lake と Fabric という大きい 2 つのプラットフォームが用意されている。2つのプラットフォームの性能は以下のようになっている。本実験では，2016 年 10 月の時点で開発がより盛んであり，主軸になっていくとされる Fabric の方を選択する。

- **Sawtooth Lake[2]**

  コンセンサスモデル:Proof of Elapsed Time, Quorum[10]

[1] https://www.hyperledger.org/

[2] http://intelledger.github.io/introduction.html

実装言語: Python

ネットワークタイプ: 非許可制，許可制

主な協力企業: intel

- Fabric[3]

  コンセンサスモデル: Batch PBFT

  実装言語: Go

  ネットワークタイプ: 許可制

  主な協力企業: IBM

2.4 Fabric

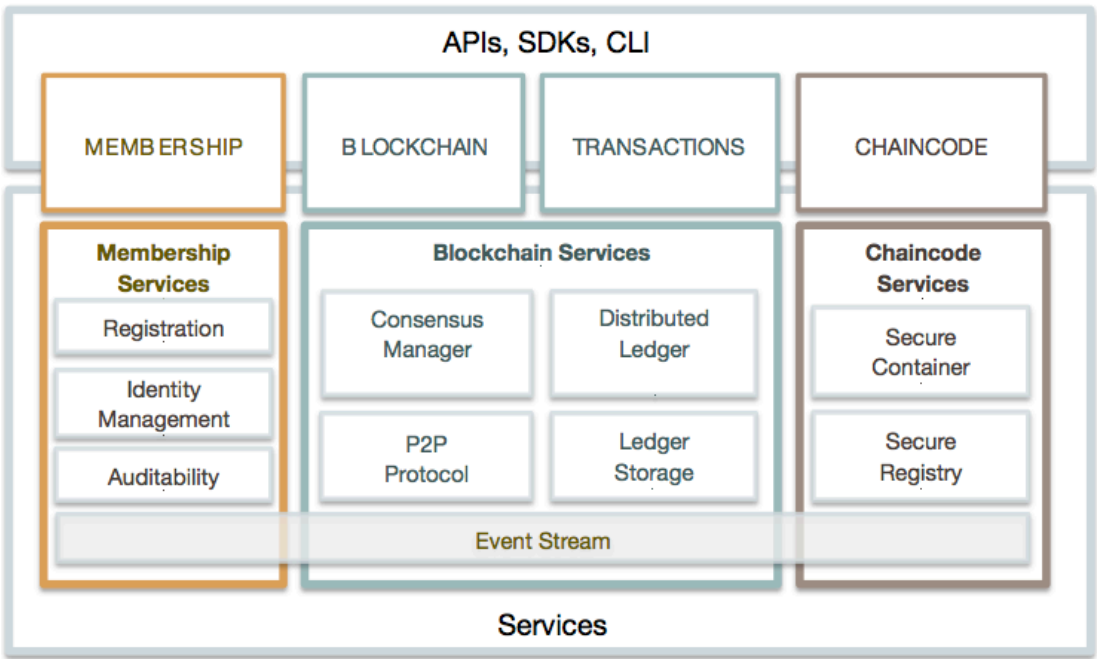

図 1 Fabric アーキテクチャ[4]

Fabric はモジュールアーキテクチャとなっており，モジュール郡から構成されたプラットフォームとなっている。図 1 はそれらをまとめた公式が提供している図であり，Fabric の全体像である。

[3] http://hyperledger-fabric.readthedocs.io/en/latest/

[4] http://hyperledger-fabric.readthedocs.io/en/latest/protocol-spec/

- **メンバーシップ**

  ネットワーク上のプライバシー，機密性，監査能力を提供している。ブロックチェーン技術の暗号技術の役割であり，PKI（Public Key Infrastructure）と分散，コンセンサスの要素を組み合わせて認可を行う。

- **ブロックチェーン，トランザクション**

  ブロックチェーン技術の台帳分散，合意形成を担う部分。コンセンサスモデルには現状では Batch PBFT のみが実装されている。今後 Raft，Proof of Work，Proof of Stake も選択可能になっていく。 通信プロトコルには Google RPC(gRPC)が利用され，台帳ストレージとしてデータストアに Rocks DB が利用されている。

- **チェーンコード**

  ブロックチェーン技術のスマートコントラクトを担う部分。セキュリティ保護されたコンテナ内で実行され，現状では Go，Java，Node.js で実装可能。

Fabric は Docker[5]で配布されており，全てのモジュールが Docker 上で実行でき，Docker が実行できる環境であれば動かすことができる。

2.5 分散データベース

分散データベース[11]とは 1 つの管理システムにより複数のマシンに接続されている記憶装置郡の形態を示す。アプリケーションからデータを見た際にはどのマシンからでも同一のデータに見える。メリットとしてはアクセスの分散による通信コストの低下，処理分散が可能な点や，一つのマシンで障害が起きても復旧可能となる障害への対応力の高さがある。つまり負荷分散と冗長化を行うことができる。一方で CAP の定理でも示されているように，一貫性，可用性，分断耐性の内同時には 2 つのみ実現可能となっている。

現在様々な分散データベースが開発され注目されている。理由には AWS や Google Cloud Platform や Azure といったクラウドサービスの普及がある。その中で特に注目されているのが，Google BigTable[12]，Amazon Dynamo[13]，BigTable をモデルとし OSS で作成された HBase，BigTable と Dynamo を参考に Facebook が開発し OSS 化された Cassandra[14]がある。

CAP の定理の観点から違いを見ると，BigTable と HBase は一貫性と分断耐性に重点が置かれているが，Dynamo と Cassandra は可用性と分断耐性に重点が置かれている。ただし Cassandra は可用性とのトレードオフで一貫性強度の選択が可能となっている。またデータの分散方法も異なる。BigTable と HBase ではデータはシャーディングされ分散されるが，Dynamo と Cassandra はコンシ

[5] https://www.docker.com/

ステントハッシュ法[15]によってデータを分散する。前者はマスタ型のアーキテクチャとなるが，後者は全ノードが同一機能を有する P2P 型のアーキテクチャとなる。

今実験では OSS であり P2P 型のアーキテクチャで，一貫性強度の選択の調整が可能であり，コンセンサスモデルが Fabric と近く，同じゴシッププロトコルが採用されている Cassandra を利用する。

3 設計と実装

3.1 概要

コンソーシアム型のネットワーク体系で資産をやり取りする分散アプリケーションを作成し，性能・実装・運用の調査をする。 作成するアプリケーションはケーススタディとして各会社がそれぞれ自社のピアを管理し，管理しているピアを通して取引を行うことを想定する。また取引を行う際には，取引の正当性と機密性を担保する仕組みを採用する。 アプリケーションは Hyperledger Project の Fabric を利用したものと，Cassandra を利用したものの 2 つを作成する。 2 つのアプリケーションとも取引は Application Programming Interface(API)で行うことができるようにし，API へ正しいリクエストが行われると処理が走るようなものになっている。また性能テストを行うために，ベンチマーカーを作成する。

3.2 環境

今研究ではピアとなるマシンを各地に立て，ネットワークを作成する必要がある。そこで Amazon Web Services(AWS)の Amazon Elastic Compute Cloud(EC2)を利用した環境構築を行う。EC2 とはクラウド内でサイズ変更が可能なコンピューティング処理能力を提供するクラウドホスティングサービスである。

全てのピアはインスタンスタイプに t2.micro(1 GiB のメモリ，1 vCPU，6 CPU)を利用し OS には Ubuntu Server 16.04 LTS (HVM), SSD Volume Type を選択している。

また AWS ではリージェンを選択することで使用するデータサーバの位置を選択することができる。ブロックチェーンは世界各地のユーザが協力して利用することが考えられる。そこでネットワークを作成する際にもネットワークの通信にかかる時間は重要となるため，考慮できるように世界各地にピアをたてネットワークを構成する。今研究では米国西部 (オレゴン)，米国東部 (バージニア北部)，EU (フランクフルト)，アジアパシフィック (東京)を利用した(図 2)。

図 2 インスタンス配置地図

3.3 共通機能

Fabric を利用したアプリケーションと Cassandra を利用したアプリケーションで共通に必要となる機能がある。今研究では会社同士が共通の目的の元，互いに取引の為に資産のやり取りをするアプリケーションとなる。この時各会社は自分の管理するマシンにのみアクセスするだけで取引と資産の確認を行う。そこで必要な機能は資産データの管理，資産の取引を行えるインターフェイス，取引を行うユーザの認証，管理である。また分散システムの面で必要な機能として，一貫性の確保などとなる。後者の分散を行うために必要な機能の多くは Fabric と Cassandra が既に持っている機能であるため，主にインターフェイス部分の実装とデータ設計をする。

3.4 Fabric

調査の章で説明した通り，Fabric はメンバーシップ，ブロックチェーン，トランザクション，チェーンコードに分かれている。メンバーシップの部分が Certificate Authority(CA)と呼ばれ，取引するユーザの登録，認証，確認や取引の正当性と機密性を担保するサービスとなっている。その為 CA は取引を行う企業ではなく共通の監視体系として一台のサーバに専用で実行させる。今後このピアを CA ピアと呼ぶ。この機能は Docker イメージで配布されているものを起動することで実行できる。起動時に読み込む設定ファイルは付録に載せる。Docker のイメージに hyperledger/fabric-membersrvc を設定し，membersrvc というコマンドで実行する。受け付けるポートは指定することができる。

次にブロックチェーン，トランザクションの設定を見ていく。Fabric ではコンセンサスモデルを No-op(コンセンサスを取らないモード)と Batch PBFT のどちらかを選択できる。1 台以上のネットワークにする際には，必ず Batch PBFT を選択することが推奨されている。この Batch PBFT を選択する際には必ずピアを 4 台以上にして設定する必要がある。設定は配布されている Docker イメージの

起動時に設定できる。今回の設定例は付録に載せる。ピアの ID は vp0 から始まり vp1,vp2 と設定していく。pki は CA つまりメンバーシップの membersvc で動作する機能であるので，動かしているピアのアドレスを指定する。CORE_PEER_DISCOVERY_ROOTNODE には構成するピアのアドレスをカンマ区切りで設定する。これにより構成するピアを認識する。CORE_PBFT_GENERAL_N はピアの数であり，CORE_PBFT_GENERAL_K は PBFT をチェックするタイミングの設定である。CORE_SECURITY_ENABLED はピアのセキュリティを CA ピアで認証するか否かであり，認証する場合は，事前に登録してある ID とキーを設定することで，構成するピアとして正しいのか認証がネットワークを作成する際に行われる。

チェーンコードはスマートコントラクトを担う部分であり，台帳上の取引の規定などを実装する。これがアプリケーションのインターフェイスの実装とデータ設計にもなる。取引を行うことが前提となっているため，普通であれば必要となるデータ設計，つまりテーブルの作成などは必要がない。このチェーンコードは現状において Go, Java, Node.js で実装可能だが，本実験では Go を利用して実装をする。現状実装される関数，つまり受け付けるリクエストの種別は NEW，EXECUTE，QUERY になっている。NEW はチェーンコードを新しく作成する関数，EXECUTE は取引が行われる関数でありつまりチェーンコードのステートを更新する関数，QUERY は状態を変更せずチェーンコードの状態の読み取りだけを行う関数である。

メンバーシップ，ブロックチェーン，トランザクション，チェーンコードからなるアプリケーションを用いることで今研究のアプリケーションを構成する。今研究で必要であった実装は，Fabric のモジュールを起動するのに必要な設定ファイルが 2 つと，インターフェイスとなる Go のチェーンコードプログラムのみとなっている。全体の構成は図 3 の通りである。図 3 は 4 社でつまり 4 つのピアで構成される想定のアプリケーションである。1 つのブロックが1つのピアを表している。

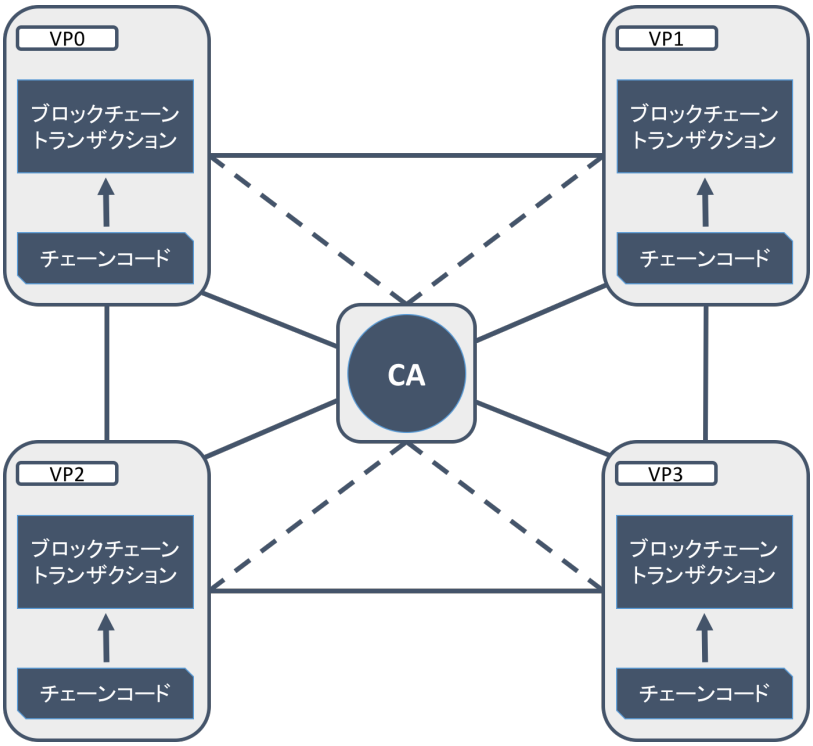

図 3 Fabric アプリケーション全体構成

各社はブロックチェーン，トランザクション，チェーンコードが実行されている 1 つのピアを管理する。この 1 つのブロックを今研究ではアプリケーションピアと呼ぶ。アプリケーションピア毎にアプリケーションのインターフェイスを持っており，リクエストがされると適切なタイミング(チェーンコードが新しく作成される時，チェーンコードのステートが更新される時，ステートを確認する時)で CA に認証の確認を問い合わせが行われるようになっている。このアプリケーションピアはアプリケーションプログラムインターフェイスを持っているため，API サーバとしても動作している。またこの API は REpresentational State Transfer(REST) API となっている。API が動作しているアドレスとポートを指定してリクエストを行うことで，取引など全ての処理を行える。行うことのできるリクエストは図 4，5，6 に示す。

NEW のリクエストでは資産の保持者である会社名と資産額を指定することで初期化が行われる。EXECUTE のリクエストでは振込を行う会社と振込先の会社名と振込額を指定することで資産のやり取りが行われる。QUERY のリクエストでは会社名を指定することでその会社の現在の資産額が返ってき確認ができる。API はデフォルトの設定では 7050 ポートで受付をしており，またこれらのリクエストは chaincode というルートに post で受け付けられる。リクエスト先 URL は http://{アドレス}:7050/chaincode のようになる。

今研究で利用した Fabric のバージョンは baseimage-v0.0.11，Go のバージョンは go1.7 linux/amd64，Docker のバージョンは 1.12.1,API version:1.24，go1.6.2 である。

```
{
  "jsonrpc": "2.0",
  "method": "deploy",
  "params": {
    "type": 1,
    "chaincodeID":{
        "name":"mycc"
    },
    "ctorMsg": {
        "args":["init", "a", "10000", "b", "10000", "c", "10000", "d",
"10000"]
    },
    "secureContext": "jim"
  },
  "id": 1
}
```

図 4 NEW リクエスト Body

```
{
  "jsonrpc": "2.0",
  "method": "invoke",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"mycc"
      },
      "ctorMsg": {
         "args":["invoke", "a", "b", "1"]
      },
      "secureContext": "jim"
  },
  "id": 2
}
```

図 5 EXECUTE リクエスト Body

```
{
  "jsonrpc": "2.0",
  "method": "query",
  "params": {
      "type": 1,
      "chaincodeID":{
          "name":"mycc"
      },
      "ctorMsg": {
         "args":["query", "a"]
      },
      "secureContext": "jim"
  },
  "id": 3
}
```

図 6 QUERY リクエスト Body

3.5 Cassandra

Cassandra が提供しているのは Fabric と異なり，データストアとしての機能とそのデータをピア間でレプリケーションを作成して同じデータとして扱えるようにする機能だけである。その為，データにアクセスするためのインターフェイスと，取引を行うリクエストに関してのその内容を解釈してデータを取り扱う必要があり，また登録されたユーザであるかの認証を行うためのプログラムを自分で実装する必要がある。つまりメンバーシップ部分である CA ピアと同様の動作をするプログラムと Fabric のチェーンコードにあたる NEW，EXECUTE，QUERY と同様の動作を行う Rest API のインターフェイスとリクエストされた内容を解釈しデータを作成，更新，管理するプログラムを実装する。今研究では，全ての実装言語は Fabric の実装でも利用されている Go を利用する。API には Gin[6] というフレームワークを利用し，データストアである Cassandra でデータを作成，更新，管理する為のクライアントに GoCQL[7]というライブラリを利用する。

Cassandra の設定は yaml 形式でファイルに記入して起動する。今回の設定は付録に載せる。設定内のアドレスは構成するピアのアドレスによって実際は異なる。Cassandra は P2P 型のアーキテ

[6] https://gin-gonic.github.io/gin/

[7] http://gocql.github.io/

クチャであり，全ノードが同等であるが，データを伝搬させるシードプロバイダーを設定しておく必要があり設定する。この役割のシードに障害が発生した場合は，新たなシードがその役割を担うことになる。

調査の章でも説明したとおり，Cassandra は一貫性のレベルを設定することができる。今研究では Fabric で設定したコンセンサスレベルと近い Quorum[5]を選択する。レプリケーション係数も選択可能であり，今研究では 2 とする。データ設計として会社の資産を扱うためのデータテーブルをキーに会社名，値に資産額となるような campany text PRIMARY KEY, money counter と作成し利用する。リクエストを受けると，EXECUTE 時には取引元には"UPDATE assets SET money = money - ? WHERE campany = ?"，取引先には"UPDATE assets SET money = money + ? WHERE campany = ?"というCQL クエリが投げられ，QUERY 時には"SELECT money FROM assets WHERE campany = ?" というような CQL クエリが投げられ，Cassandra にアクセスする。

Fabric の CA ピアと同様の機能をもつためのアプリケーションは，ユーザデータの管理とリクエストを受けると管理されたデータを参照に認証を行う機能を実装している。問い合わせを受け付け，レスポンスを行うためのインターフェイスを持つ。

アプリケーション全体の構成は図 7 の通りである。図の通り Fabric を用いたアプリケーションに近い構成になる。Fabric と同様にアプリケーションピアは適切なタイミングで CA ピアに認証の確認の問い合わせが行われるようになっている。作成したアプリケーションピアの Go 製の API は図 8 のようになっている。利用した JAVA のバージョンは 1.8.0_111, Runtime Environment (build 1.8.0_111-8u111-b14-2ubuntu0.16.04.2-b14, 64-Bit Server VM (build 25.111-b14, mixed mode), Go のバージョンは go1.7 linux/amd64, Cassandra のバージョンは 3.6 である。

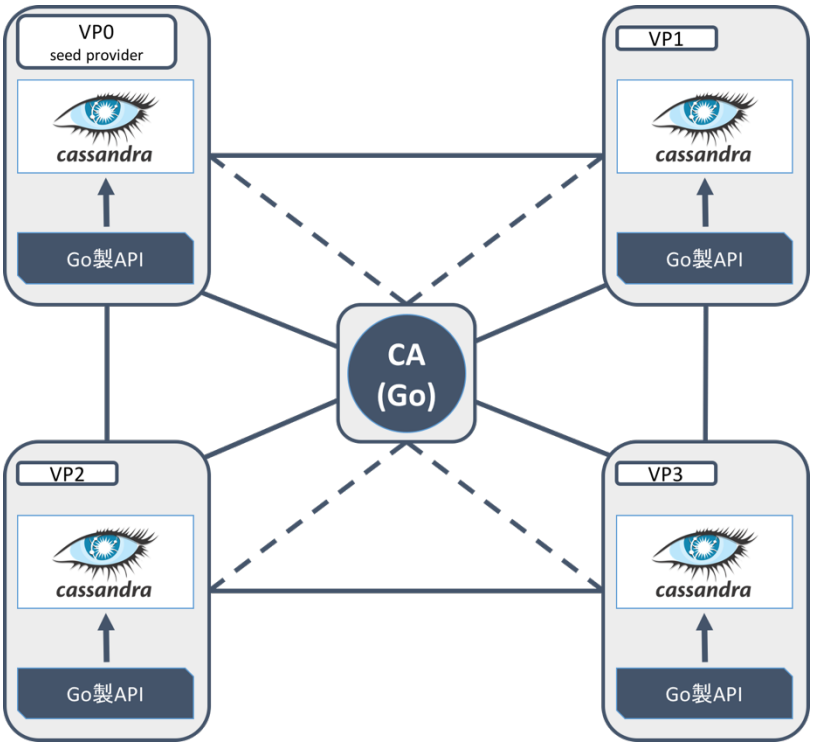

図 7 Cassandra アプリケーション全体構成

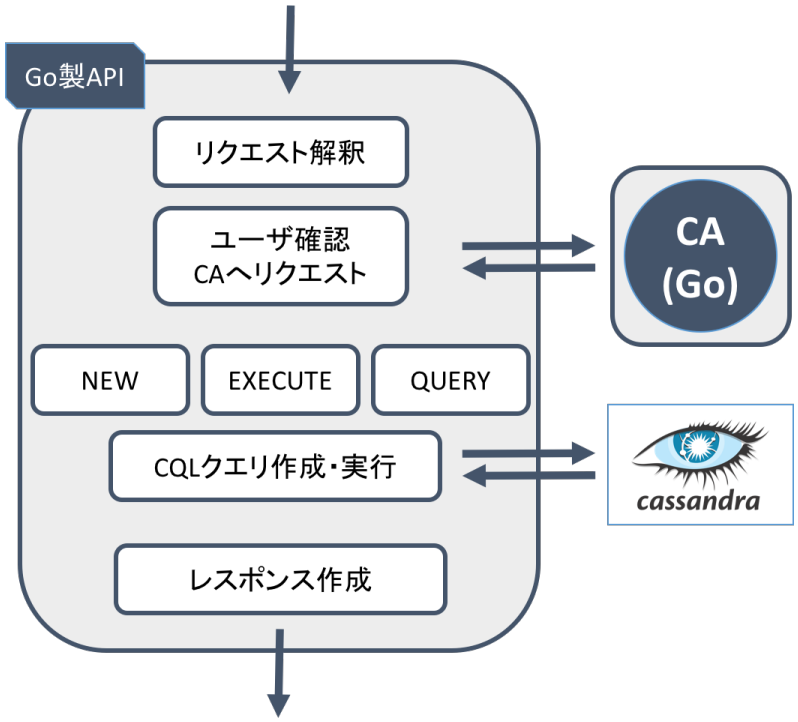

図 8 Cassandra を用いたアプリケーションの Go 製 API

3.6 ベンチマーカー

2つのアプリケーションの性能評価を行うためにベンチマーカーを作成する。ベンチマーカーは NEW，EXECUTE，QUERY のパラメータを JSON Body に持つ http リクエストを一定時間送る機能を持つ。実行の際には指定したアドレスとベンチマークの実行時間とワーカ数を指定する。アドレスは複数指定可能であり，複数指定した場合は一つのアドレス毎に 1 つのスレッドが走り並列に http リクエストが行われる。ワーカ数とは並列にリクエストを送る数であり，通常 1 アドレス 1 スレッドの並列度だが，これを 1 アドレスに指定した数のスレッドの並列度にすることができる。

計測はリクエスト毎にリクエストを出してから返ってくるまでの時間つまりリクエスト毎のレスポンスタイムを測定する。またリクエストの成功回数と失敗回数をカウントしていく。リクエスト毎に全てのリクエスト時間を保存し，最後に CSV 形式に出力する。この時最もかかったリクエスト時間と平均時間も出力する。 実装は言語に Go を利用している。並列化には Go の機能である goroutine を利用する。goroutine の仕組みは軽量なスレッドのようなもので，独立に実行する複数の関数(コルーチン)を，スレッドの集合に多重化するものとなっている[16]。 実装に利用した Go のバージョンは go1.7.1 darwin/amd64 である。

4 性能評価

4.1 概要

実装した2つのアプリケーションの性能評価を実装したベンチマーカーで行う。性能評価はピアの数を変えたいくつかの構成で行う。今研究で行った構成，評価方法を説明し，行った構成毎のアプリケーションの評価結果をまとめる。

4.2 構成

性能評価を行ったネットワーク構成は全 3 つである。ピアの場所は AWS 準拠で表記する。ケース 1 はアジアパシフィック（東京)の Certificate Authority となるピア 1 つと米国西部（オレゴン)のアプリケーションピア 1 つで構成されるネットワーク。ケース 2 はアジアパシフィック（東京)の Certificate Authority となるピアと米国西部（オレゴン)，米国東部（バージニア北部)，EU（フランクフルト)，アジアパシフィック（東京)に 1 つずつのアプリケーションピアで構成されるネットワーク。ケース 3 はアジアパシフィック（東京)の Certificate Authority となるピアと米国西部（オレゴン)，米国東部（バージニア北部)，EU（フランクフルト)，アジアパシフィック（東京)に 2 つずつのアプリケーションピアで構成されるネットワーク。全てのピアは設計の章の環境で書いてあるとおりである。

全てのケースにおいてベンチマーカーは手元の PC(OS X Yosemite version 10.10.5 8 GB 1600 MHz DDR3 東京)から実行する。

4.3 評価方法

各アプリケーションピアは基本的に一つの会社が所有しているものとする。その為ピアの数だけ企業が存在する状態になる。ただしケース 1 の場合は 1 つのアプリケーションを 2 つの企業で利用していることを想定した運用とする。

初期状態において全ての会社は 10000 の資金を持つ。この初期化の処理はベンチマークを行う前にする。

ベンチマーカーは各アプリケーションピアに実装の章で説明した EXECUTE のリクエストを行う。EXECUTE の内容はリクエスト先のピアを所有する自身の会社名と取引先の会社名と取引額(今評価実験では常に 1)とログイン済みの自身を表すユーザ証明済みの名前からなるリクエストを行う。リクエスト 1 つで 1 つの取引が行われることになる。図 7 のリクエスト内容であれば a 社から b 社へ 1 の資金を送金したことになる。

ベンチマーカーは各アプリケーションピアに指定したワーカ数で 60 秒間リクエストを行い続ける。評価結果は成功数，失敗数，最大レスポンス時間，最小レスポンス時間，レスポンスタイム平均値，レスポンスタイム中央値，レスポンス標準偏差とレスポンスタイムを時系列にグラフ化したものと，0.05s 毎にまとめ頻出度をグラフ化したものを載せる。

4.4 評価結果

- **ケース 1**

  アプリケーションピア: オレゴン 1 つ

  CA ピア: 東京

  ベンチマーカー: 東京

表 1 ケース 1 評価結果

	worker1		worker4		worker8	
	Fabric	Cassandra	Fabric	Cassandra	Fabric	Cassandra
成功数	280	266	799	1019	869	2052
失敗数	0	0	0	0	0	0
最大時間(s)	0.807	0.458	0.851	0.479	2.842	0.481
最小時間(s)	0.139	0.223	0.145	0.222	0.139	0.218
平均値(s)	0.215	0.226	0.301	0.236	0.555	0.234
中央値(s)	0.217	0.225	0.267	0.235	0.516	0.230
標準偏差(s)	0.049	0.014	0.131	0.016	0.329	0.020

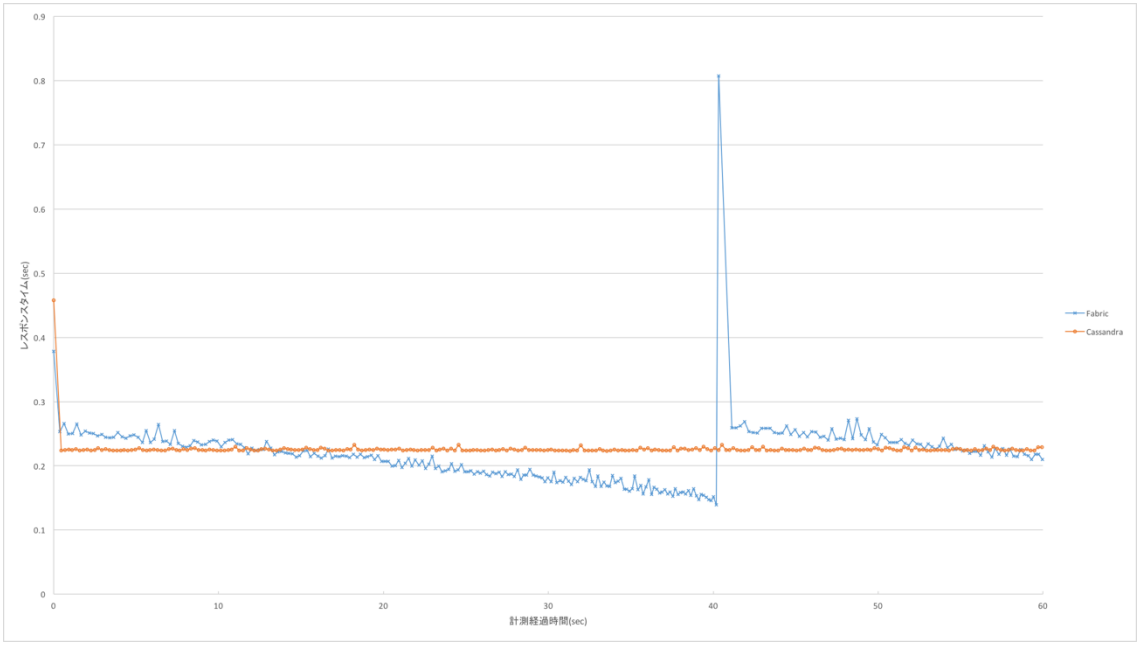

図 9 ケース 1worker1 時系列レスポンスタイムグラフ

図 10 ケース 1worker1 レスポンスタイム頻出グラフ

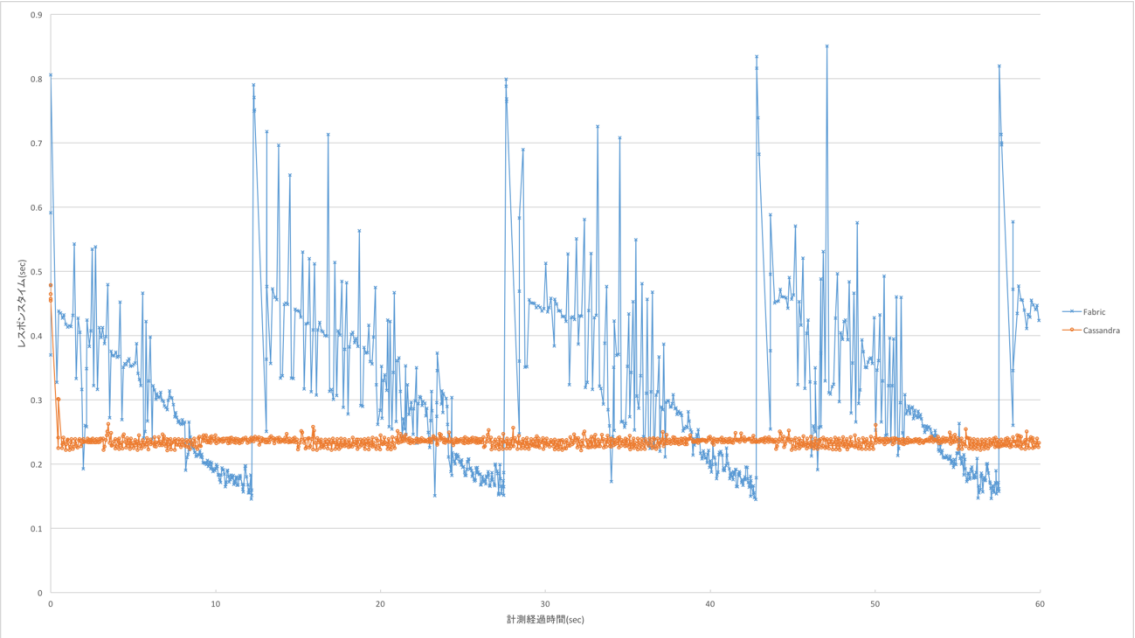

図 11 ケース 1worker4 時系列レスポンスタイムグラフ

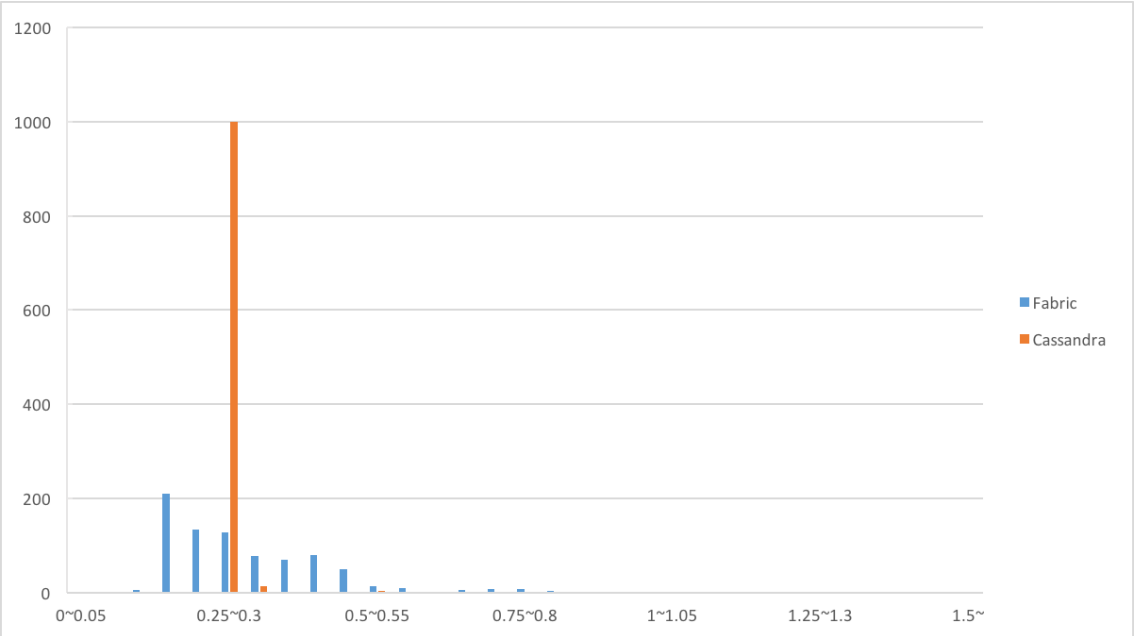

図 12 ケース 1worker4 レスポンスタイム頻出グラフ

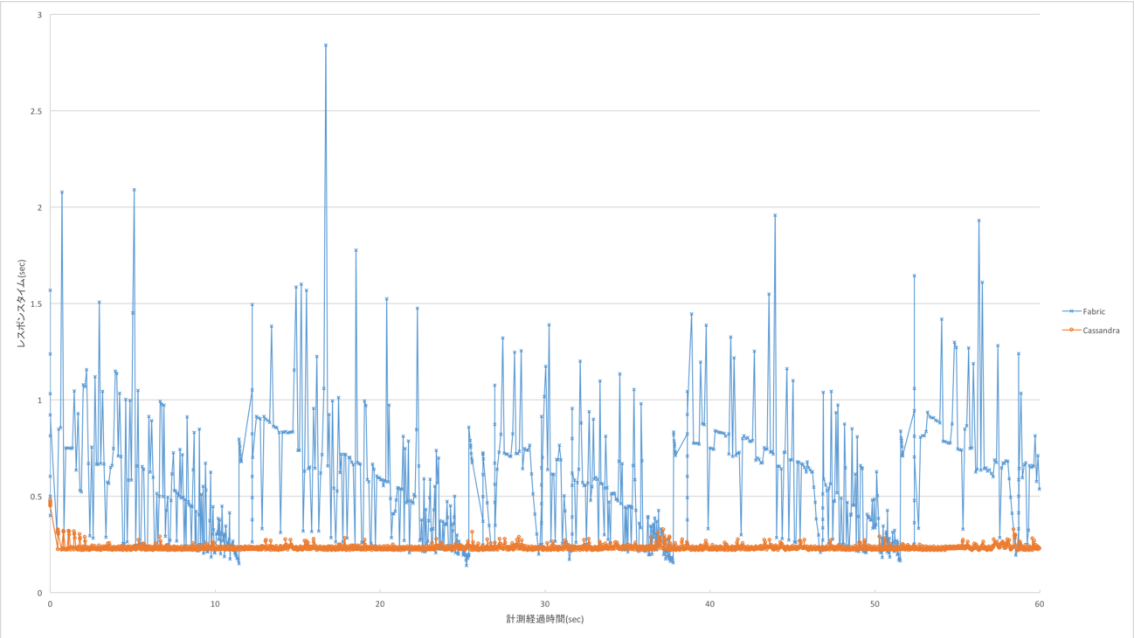

図 13 ケース 1worker8 時系列レスポンスタイムグラフ

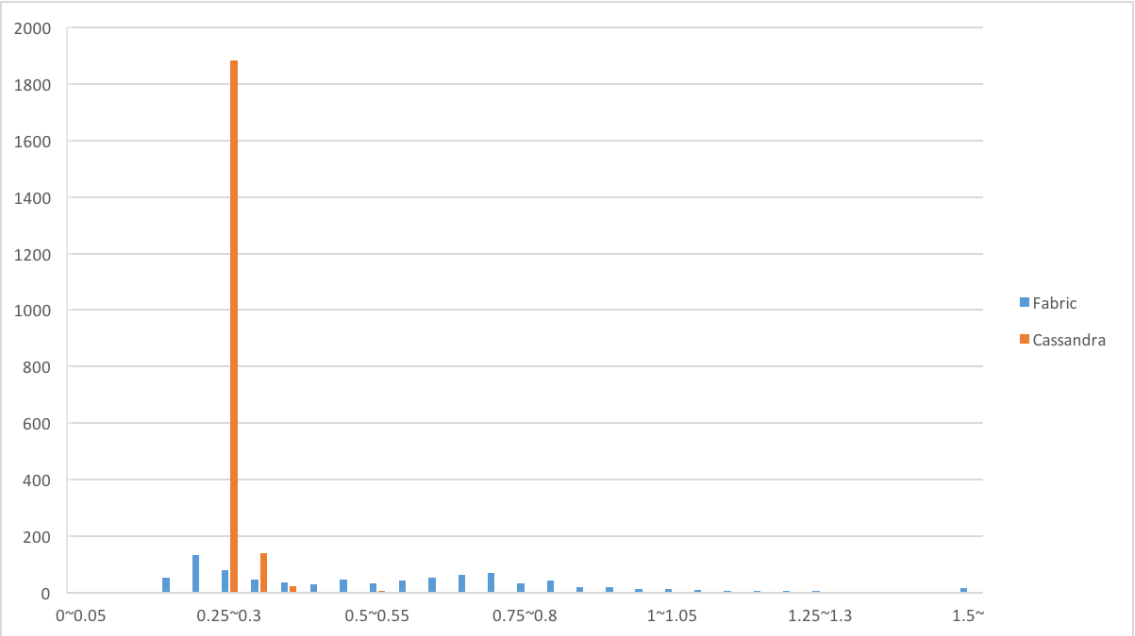

図 14 ケース 1worker8 頻出グラフ

- **ケース 2**

  アプリケーションピア: オレゴン(a)，フランクフルト(b)，バージニア北部(c)，東京(d)

  CA ピア: 東京

  ベンチマーカー: 東京

  このケースでベンチマークが終わった後にアプリケーションピアで処理が動き続け処理が終わらない限り正しい資産値を取得できない現象が起きた。ベンチマーク終了後からこの処理にかかった時間をバッチラグとする。

表 2 ケース 2 評価結果

	worker1		worker4		worker8	
	Fabric	Cassandra	Fabric	Cassandra	Fabric	Cassandra
成功数	718	223	1064	925	3525	1855
失敗数	0	27	0	89	0	210
最大時間(s)	3.714	2.001	11.985	2.300	8.212	2.295
最小時間(s)	0.045	0.429	0.056	0.420	0.042	0.399
平均値(s)	0.335	0.976	0.933	0.955	0.548	0.938
中央値(s)	0.251	0.919	0.406	0.917	0.346	0.908
標準偏差(s)	0.323	0.275	1.135	0.258	0.550	0.268
バッチラグ(s)	0	-	0	-	140	-

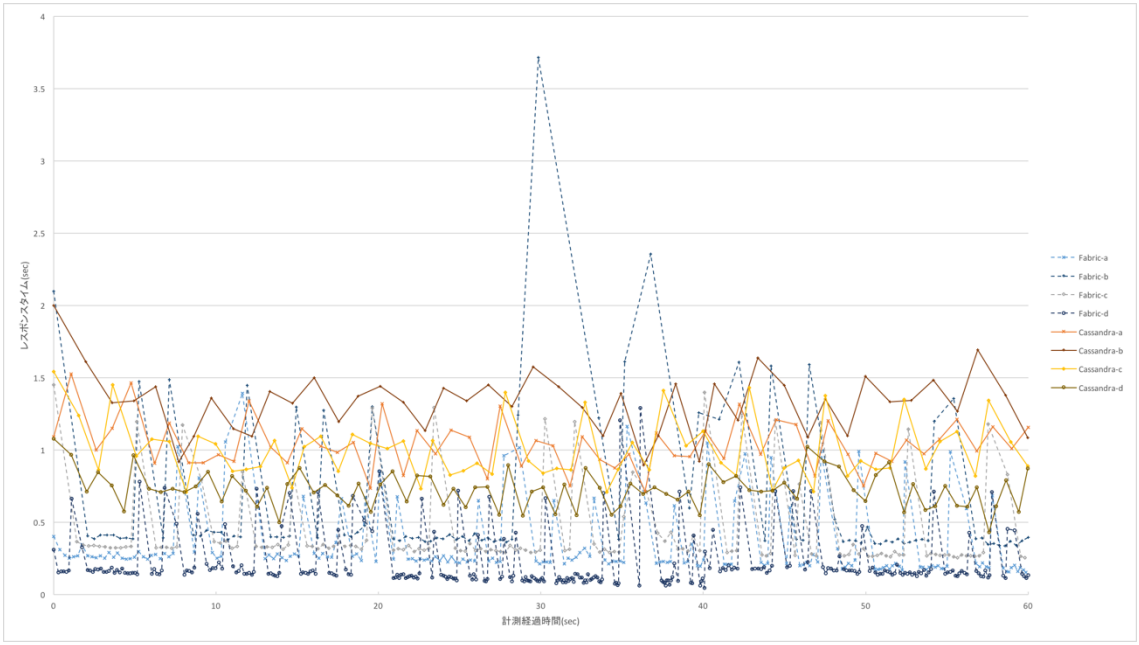

図 15 ケース 2worker1 時系列レスポンスタイムグラフ

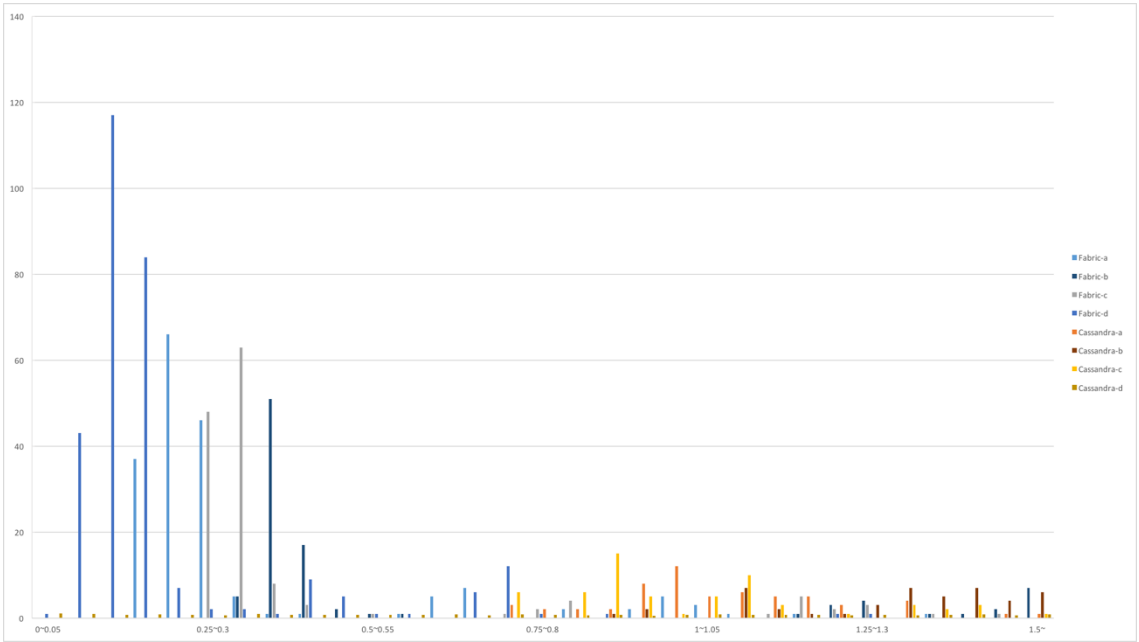

図 16 ケース 2work1 レスポンスタイム頻出グラフ

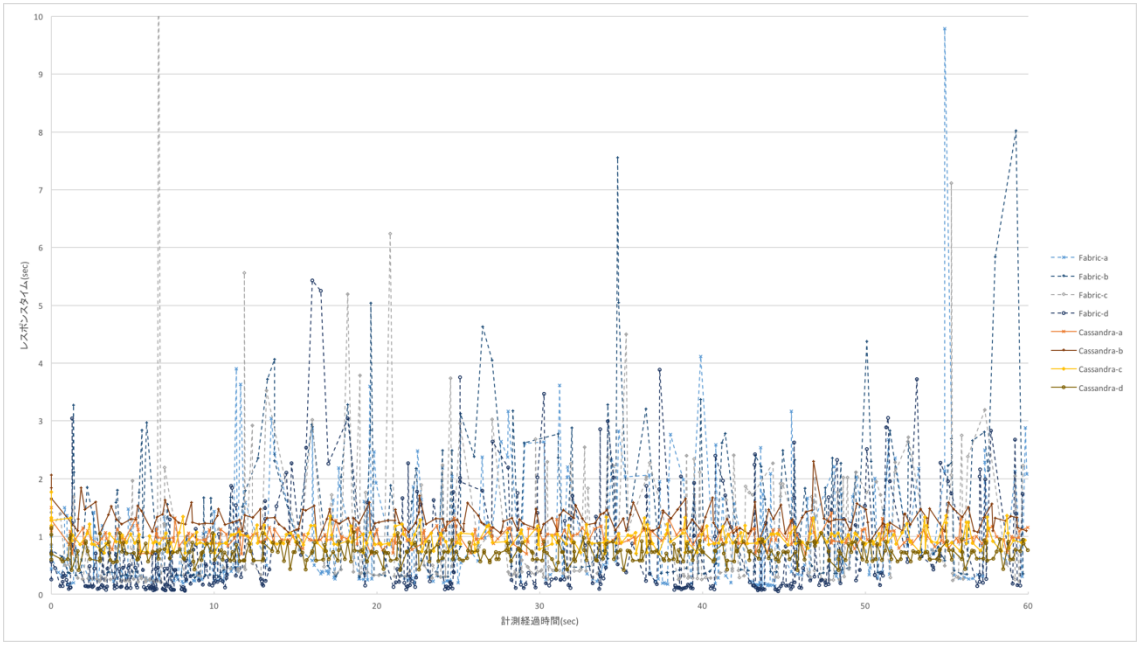

図 17 ケース 2worker4 時系列レスポンスタイムグラフ

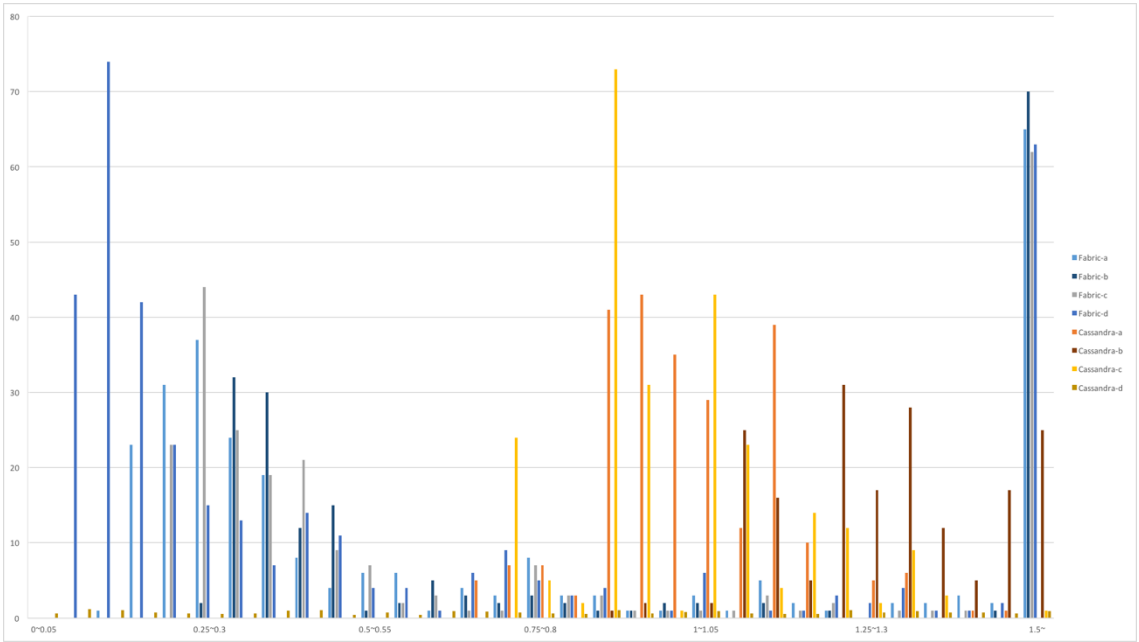

図 18 ケース 2worker4 レスポンスタイム頻出グラフ

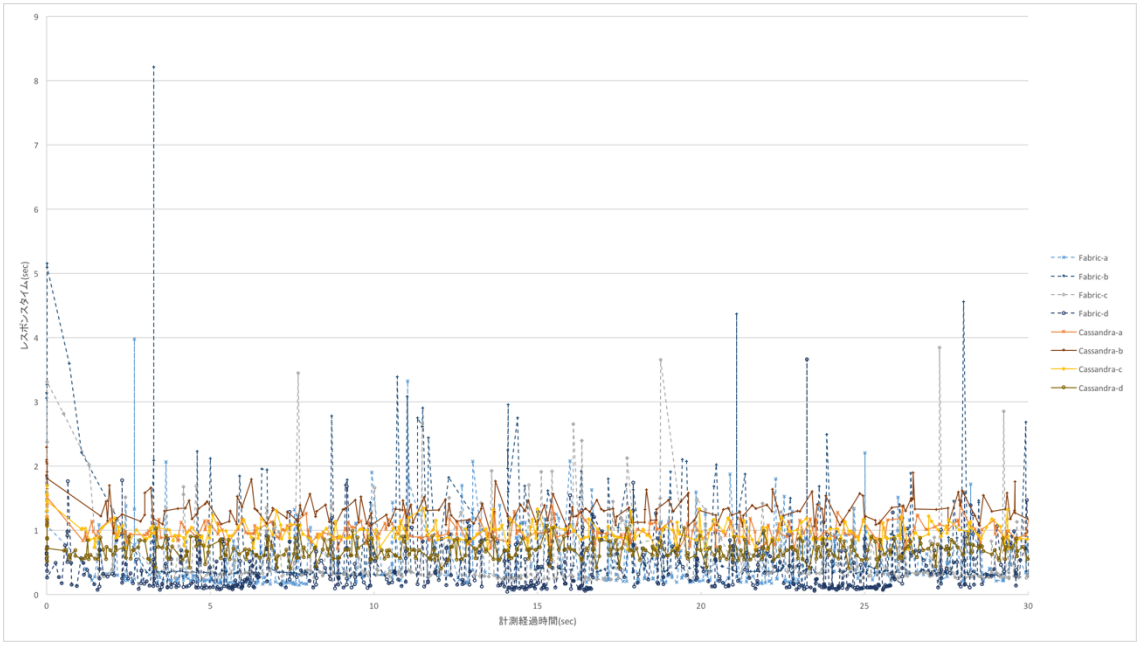

図 19 ケース 2work8 時系列レスポンスタイムグラフ(0~30)

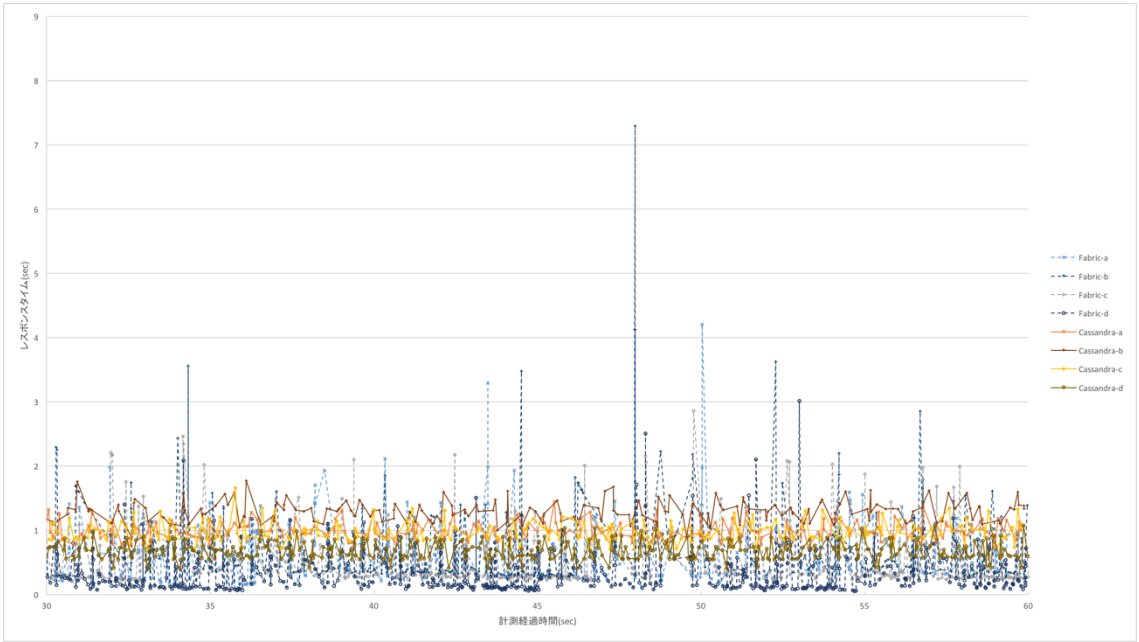

図 20 ケース 2worker8 時系列レスポンスタイムグラフ(30~60)

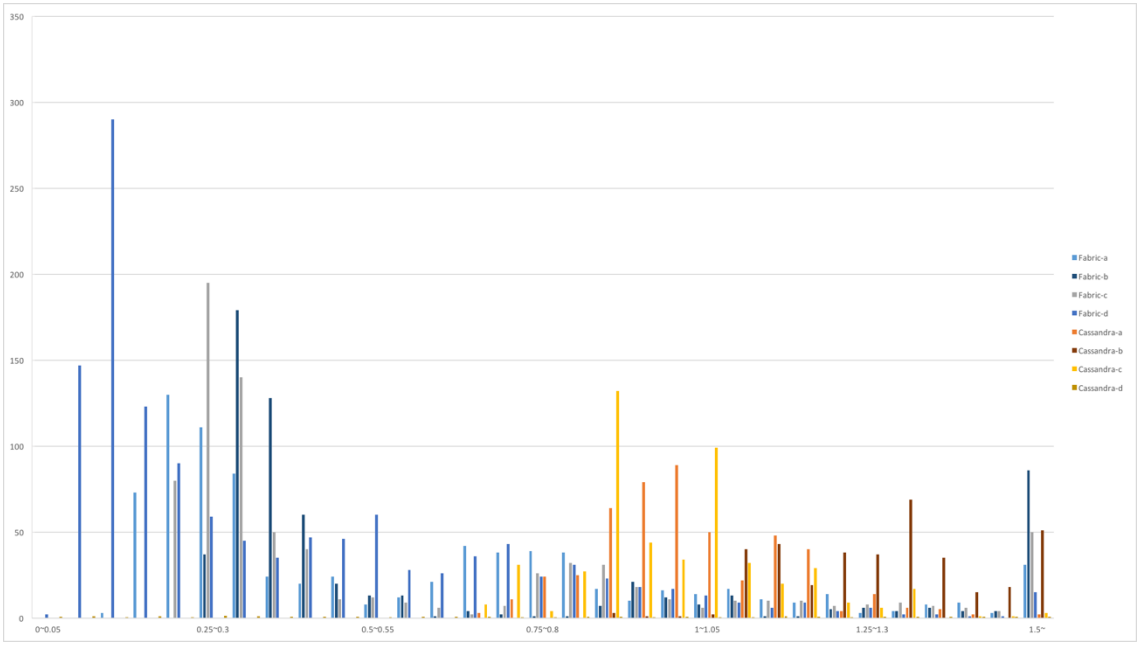

図 21 ケース 2worker8 レスポンスタイム頻出グラフ

- ケース 3

アプリケーションピア: オレゴン(a,e), フランクフルト(b,f), バージニア北部(c,g), 東京(e,h)

CA ピア: 東京

このケースでは Fabric を利用したアプリケーションと Cassandra を利用したアプリケーションで同じワーカ数で一定時間内のリクエスト総数に大きな差が出た。リクエスト総数が近くなるようにリクエストに 50ms 間隔をおいて worker(50ms)をし, Cassandra には多く行った。

表 3 ケース 3 評価結果

	worker1		worker2	worker4	
	Fabric	Cassandra	Fabric	Fabric	Cassandra
成功数	2605	515	4742	8313	2084
失敗数	0	38	0	0	151
最大時間(s)	1.648	1.973	1.680	1.735	2.064
最小時間(s)	0.030	0.310	0.028	0.031	0.308
平均値(s)	0.185	0.875	0.203	0.232	0.866
中央値(s)	0.171	0.838	0.193	0.214	0.828
標準偏差(s)	0.107	0.282	0.130	0.166	0.274
バッチラグ(s)	32	-	267	1003	-

表 3 ケース 3 評価結果

	worker(50ms)	worker8	worker16	worker32	worker64
	Fabric	Cassandra			
成功数	1911	4093	8202	16538	32975
失敗数	0	313	650	1388	2826
最大時間(s)	1.660	2.485	2.285	2.264	2.235
最小時間(s)	0.033	0.308	0.307	0.124	0.065
平均値(s)	0.200	0.878	0.874	0.864	0.864
中央値(s)	0.201	0.855	0.853	0.837	0.841
標準偏差(s)	0.111	0.281	0.279	0.278	0.275
バッチラグ(s)	0	–	–	–	–

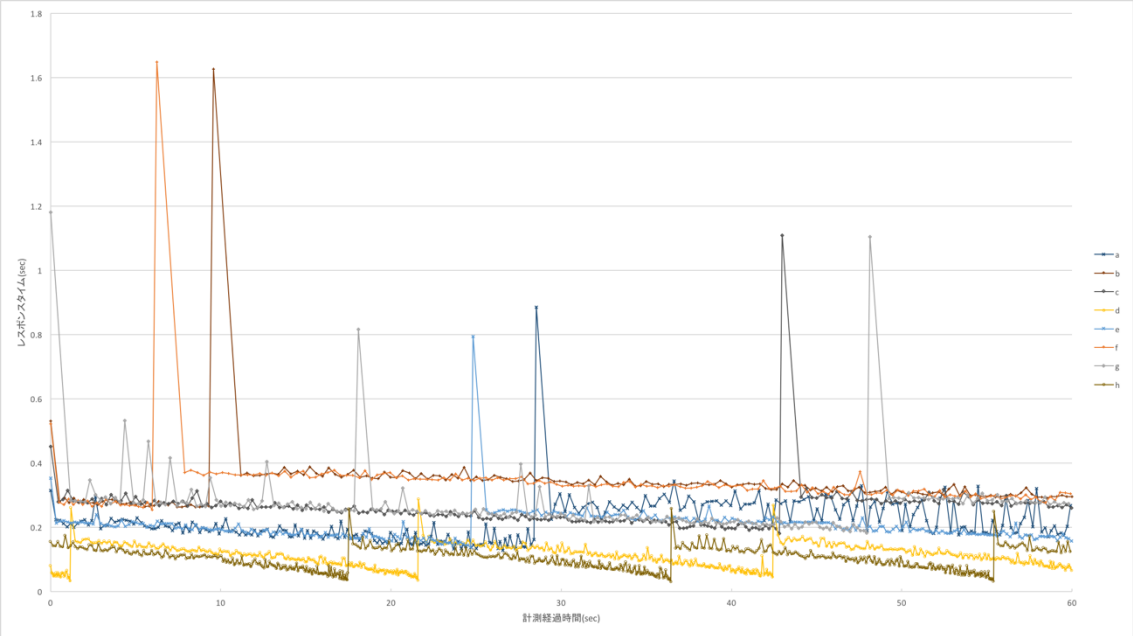

図 22 ケース 3worker1Fabric 時系列レスポンスタイムグラフ

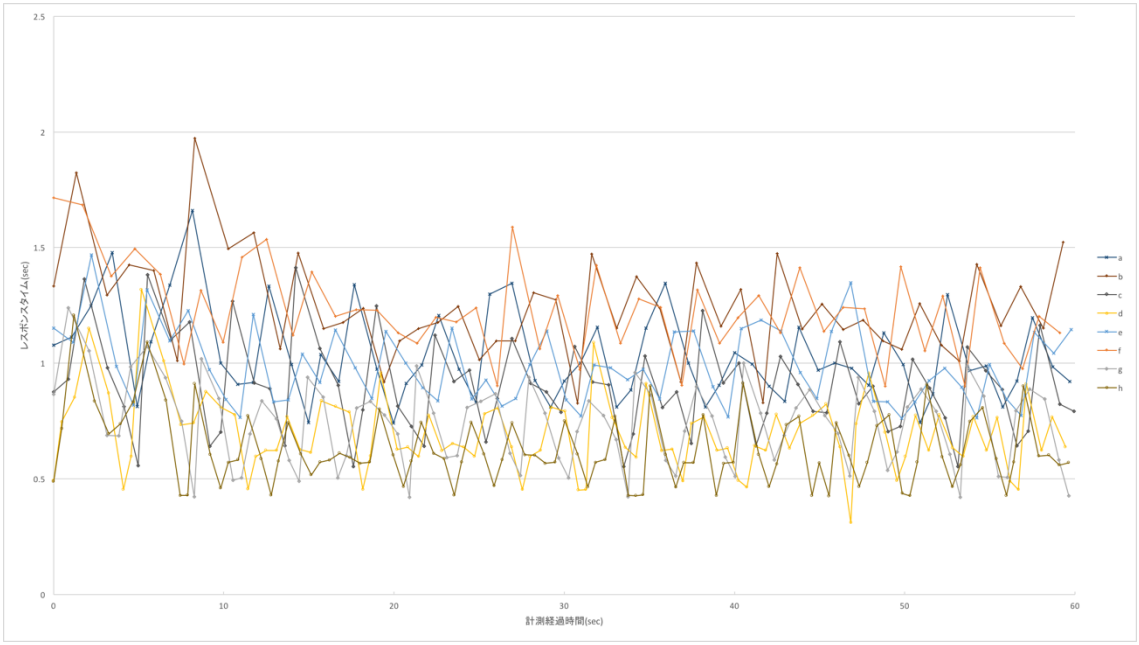

図 23 ケース 3worker1Cassandra 時系列レスポンスタイムグラフ

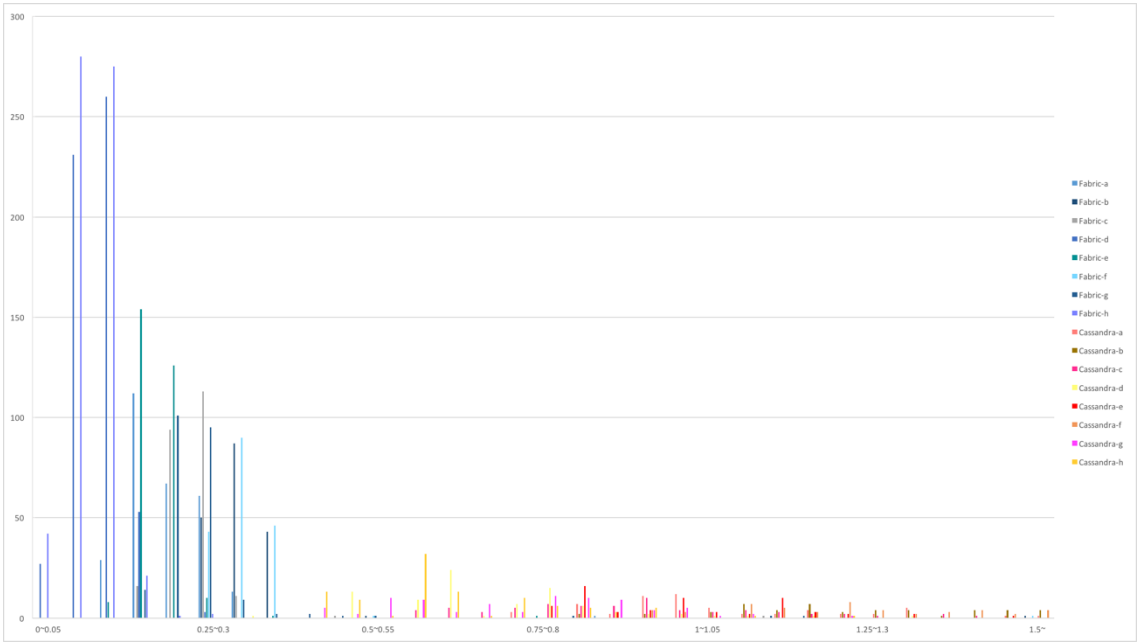

図 24 ケース 3worker1 レスポンスタイム頻出グラフ

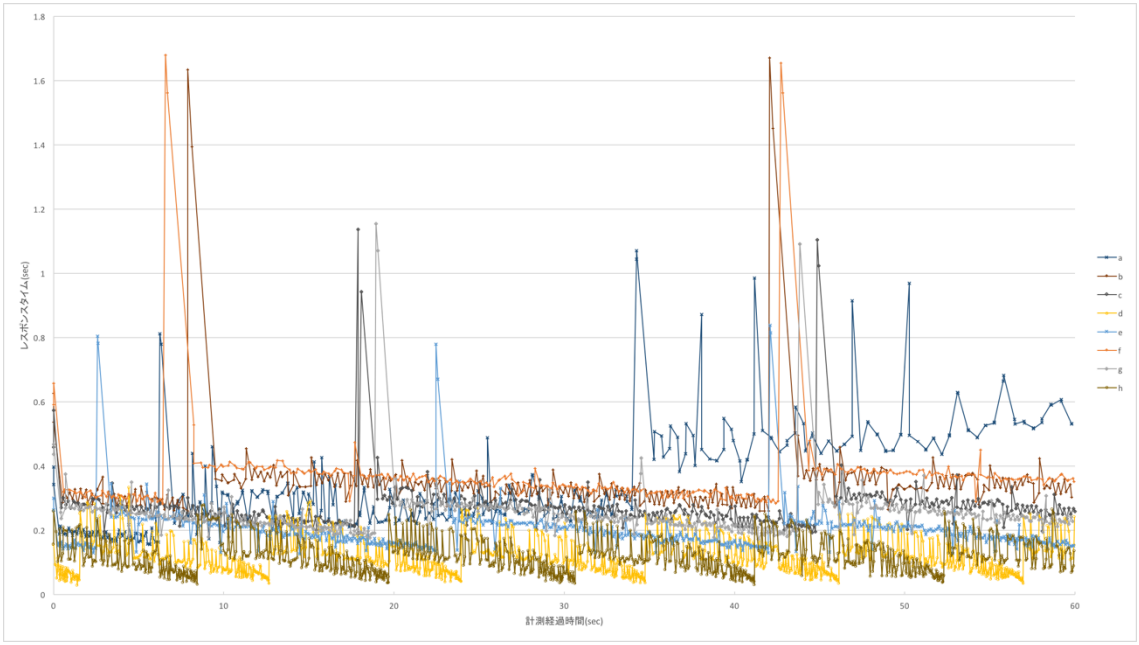

図 25 ケース 3worker2Fabric 時系列レスポンスタイムグラフ

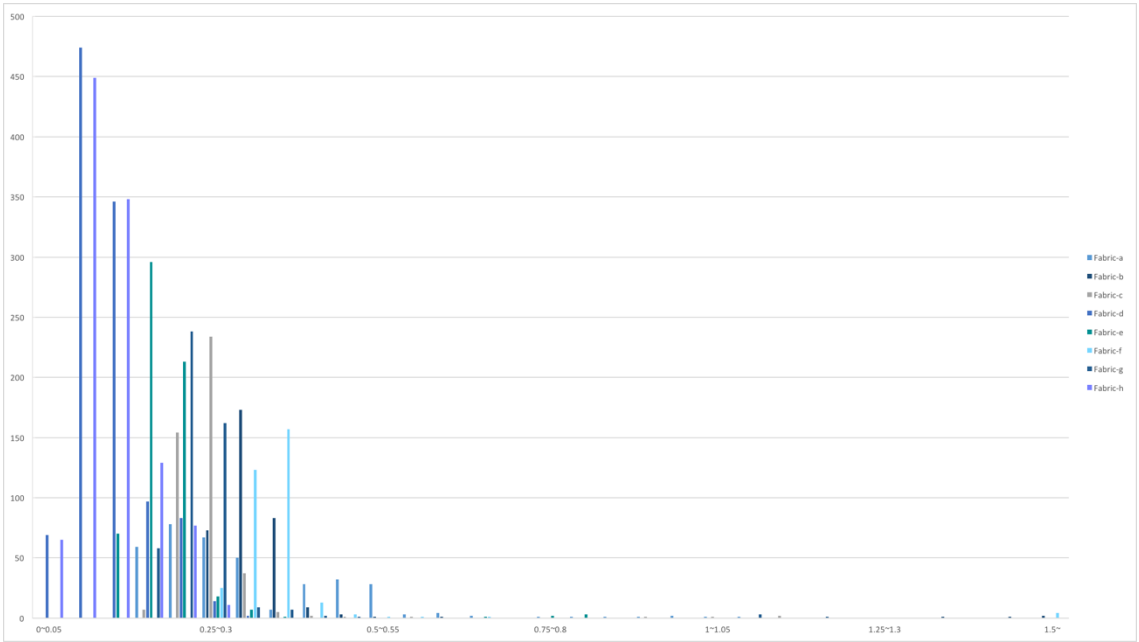

図 26 ケース 3worker2Fabric レスポンスタイム頻出グラフ

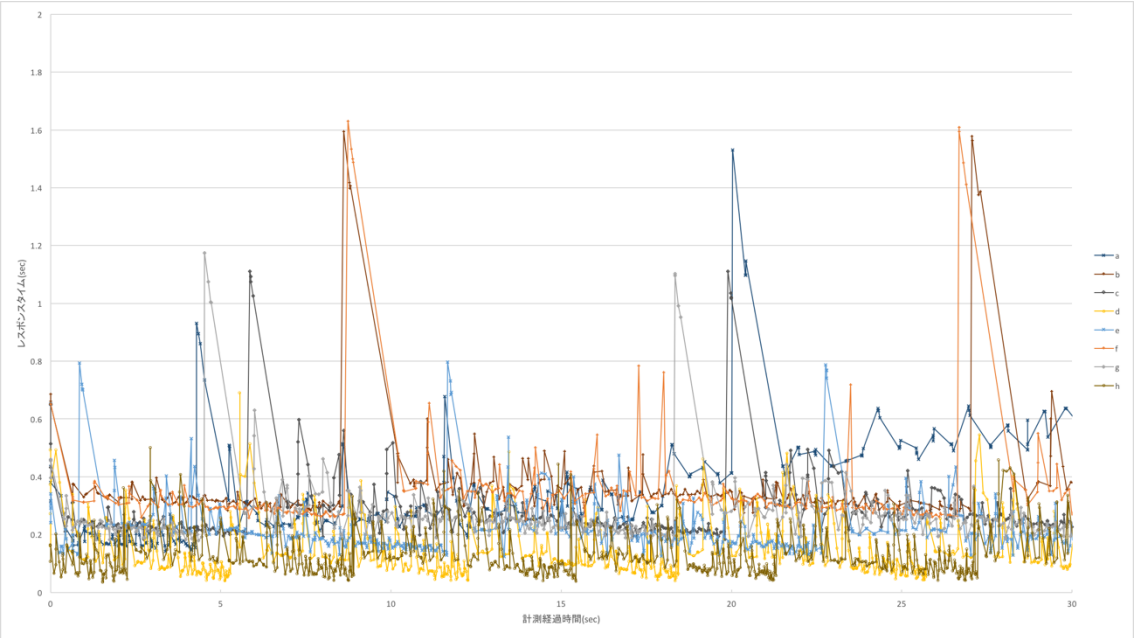

図 27 ケース 3worker4Fabric 時系列レスポンスタイムグラフ(0~30)

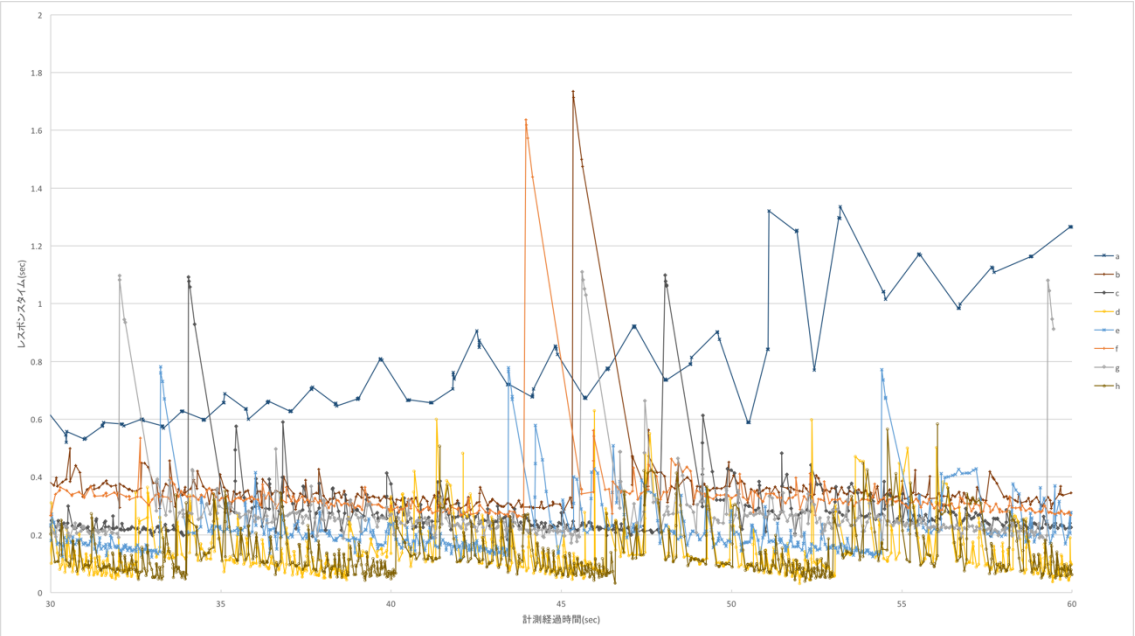

図 28 ケース 3worker4Fabric 時系列レスポンスタイムグラフ(30~60)

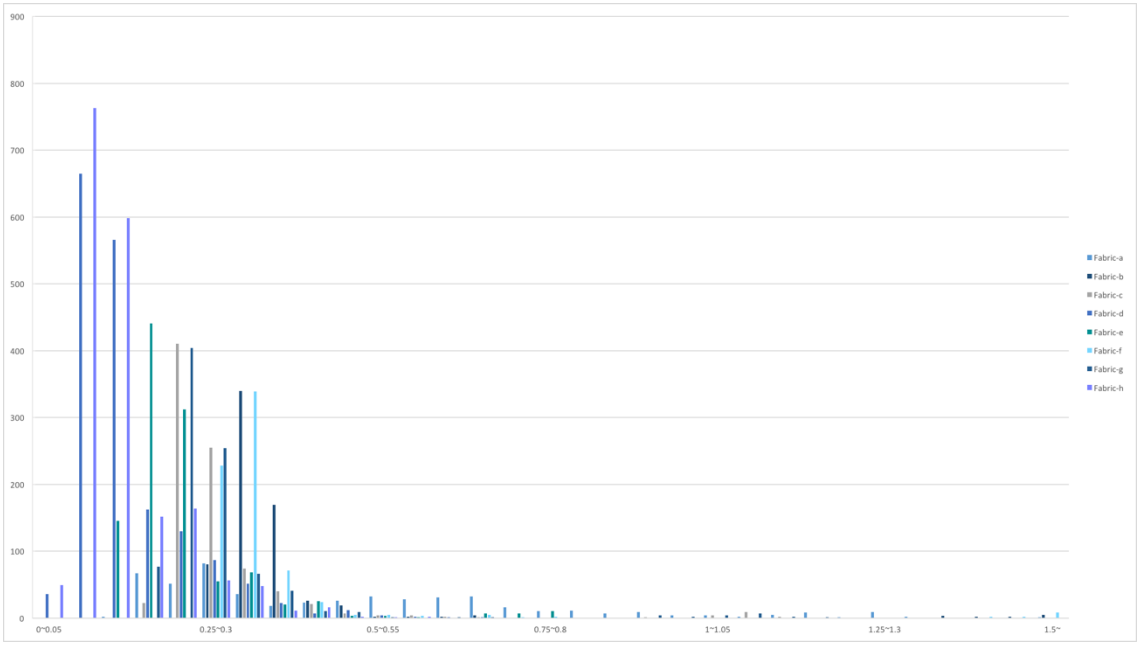

図 29 ケース 3worker4Fabric レスポンスタイム頻出グラフ

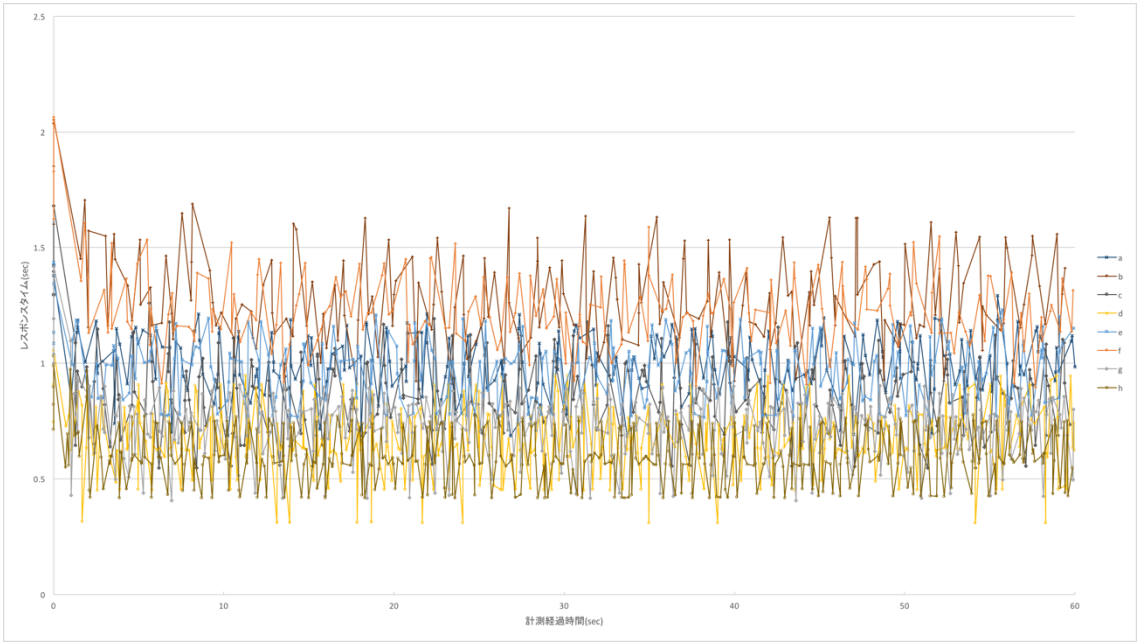

図 30 ケース 3worker4Cassandra 時系列レスポンスタイムグラフ

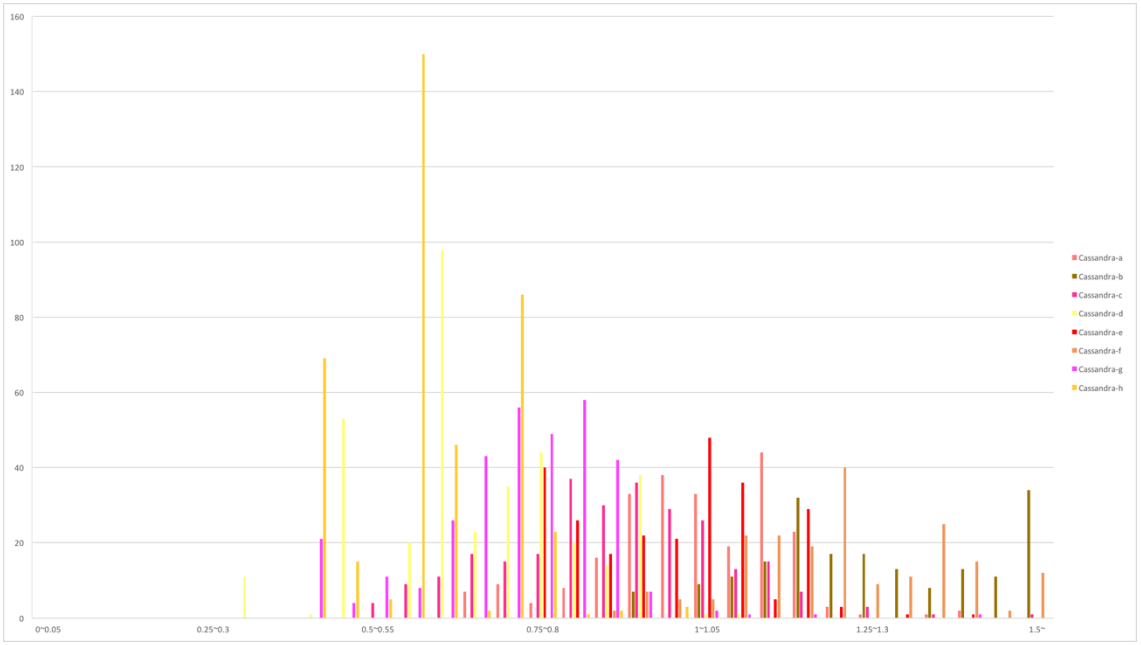

図 31 ケース 3worker4Cassandra レスポンスタイム頻出グラフ

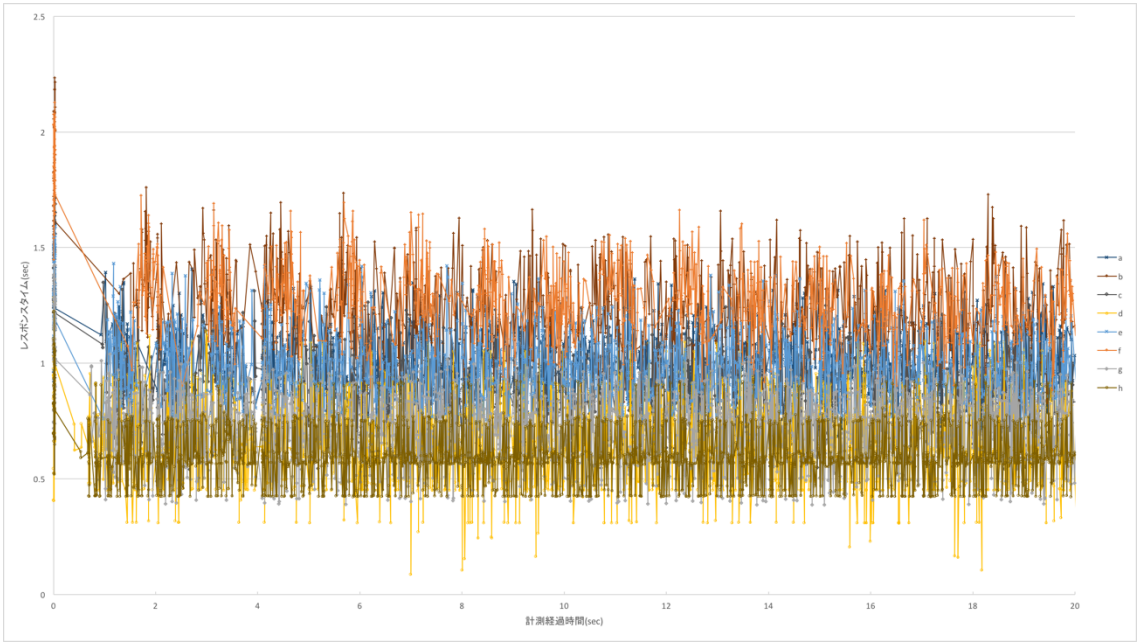

図 32 ケース 3worker64Cassandra 時系列レスポンスタイムグラフ(0~20)

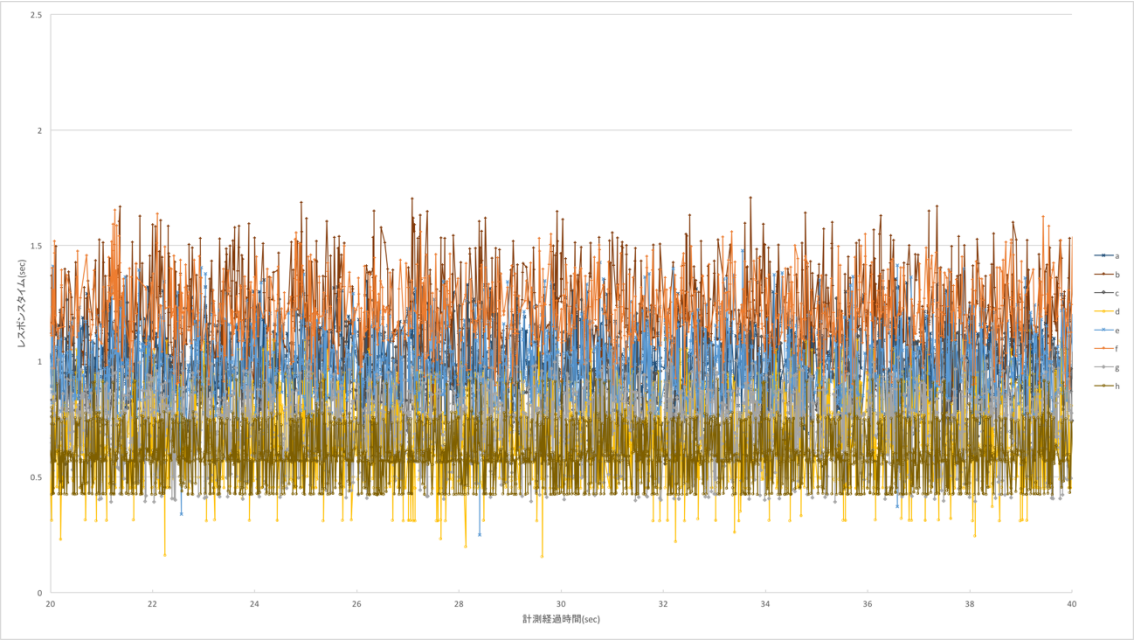

図 33 ケース 3worker64Cassandra 時系列レスポンスタイムグラフ(20~40)

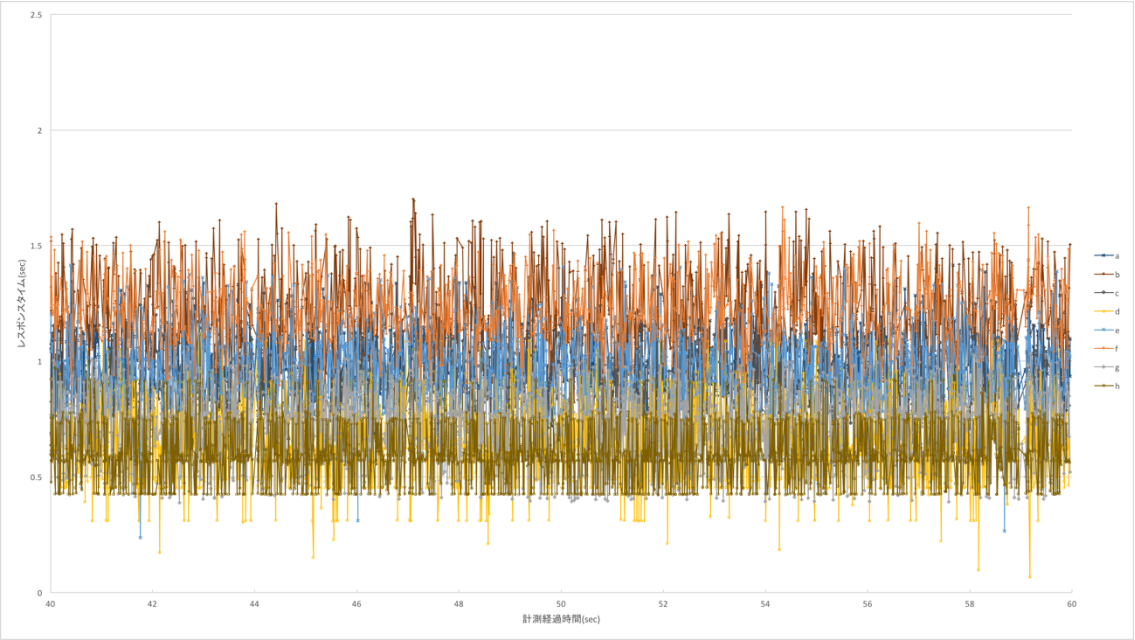

図 34 ケース 3worker64Cassandra 時系列レスポンスタイムグラフ(40~60)

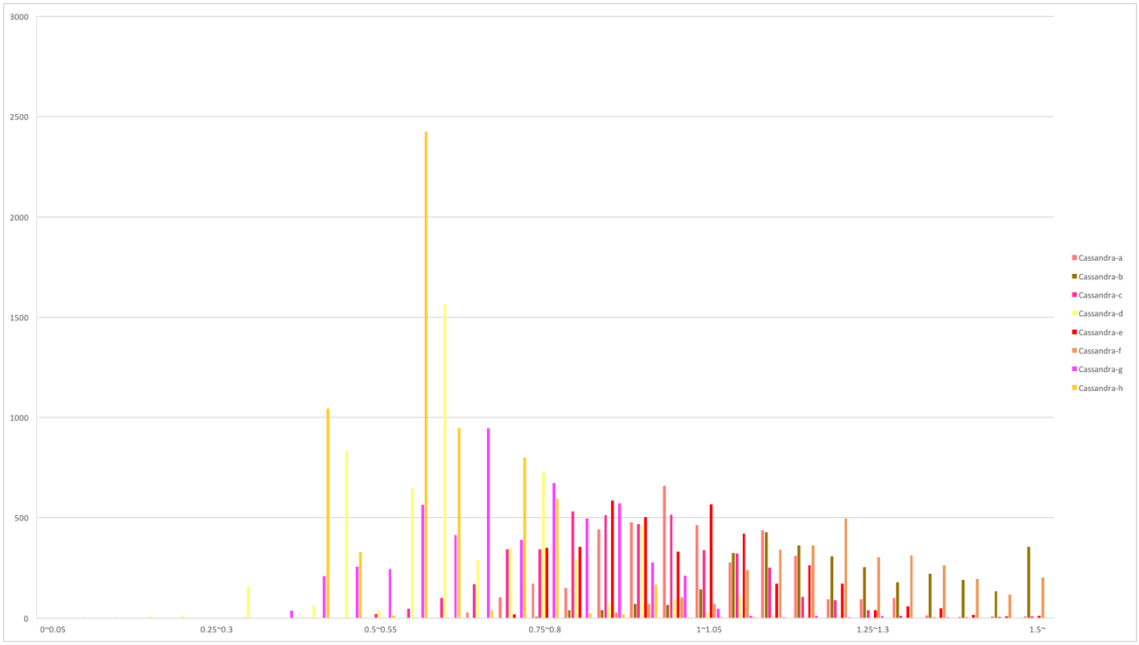

図 35 ケース 3worker64Cassandra レスポンスタイム頻出グラフ

5 考察

評価実験の結果の考察と，評価実験を行うために実装したアプリケーションの実装コスト，可用性，冗長性，柔軟性などの運用の考察を行う。

5.1 評価結果

まずケース 1 の，ピアを1つだけの構成であり，分散されていない単体の性能評価結果を見る。Cassandra のレスポンスタイムはほぼ一定になっているのに対して，Fabric のレスポンスタイムはある程度時間が立つとレスポンスタイムが跳ね上がっている点があることがわかる。Fabric では取引が行われる毎にブロックつまりデータが増えていく。一定の取引が行われると Fabric が内部的に利用しているデータストアである Rocks DB へと書き込みが走る。また Rocks DB インメモリに乗り切らないと永続化のトランザクション処理が走る。これは Cassandra でも同様であるが Cassandra のデータ設計ではデータ量は増えず,Fabric のようなブロックチェーンではそのデータ構造上，取引を行うほどにデータ量が増えていくためこのような差が発生した考えられる。またワーカ数を 8 にした際に Fabric の方ではレスポンスタイムが平均値，中央値共に上昇したことから，Fabric のアプリケーションのデータの更新自体が並列処理に弱く，データの更新の速度が遅いと考えられる。これはそも

そも更新時に書き込む情報量が多いことから，一つ一つの更新に時間がかかり，データの書き込みの間ロックされる時間が長いために起きているのではないかと考えられる。

次にケース 2 のピア 4 つでネットワークが構成されているアプリケーションの性能評価結果を見る。ケース 1 の時と同様に標準偏差が Cassandra より Fabric が大きい。その一方で，ケース 1 の際には平均値，中央値共に Fabric と Cassandra にそこまでの差がなかったが，ケース 2 では Fabric のほうが平均値と特に中央値が小さくなっている。これらはこの時の頻出グラフからもわかる。またこの時の時系列グラフから Fabric はレスポンスタイムが短く返ってきているが時折レスポンスタイムが長くなり，また突出してレスポンスタイムのかかる時もあることがわかる。時折レスポンスタイムが長くなるのはケース 1 のときと同様の現象であり，データストアへの書き込みやデータストアのトランザクション処理が走った事によると考えられる。

また Cassandra は全体的にケース 1 の特に比べレスポンスタイムが伸びているが，Fabric はケース 1 のときと同様のレスポンスタイムで多くが返ってきている。これは一貫性をとるコンセンサス方法とデータ分散の違いが原因であると考えられる。Cassandra ではどのピアにデータがあるかハッシュ表を元に見つけ出し書き込みが行われる。 そしてこの際に一貫性を QUORUM のレベルで確保するようにレプリケーションを取っているピアと確認をする。一方で Fabric では全てのピアで全てのデータを持つため，まずリクエストを受け付け，その後にバッチ処理で一貫性を担保する。その為ケース 1 に比べて Cassandra では 0.2s ほどレスポンスタイムが伸びているが，Fabric では Batch PBFT というバッチ処理で行うという方式がとられている為，取引を行うリクエスト毎に一貫性をとる処理が走るわけではなく，ケース 1 の時に近いレスポンスタイムで多くのリクエストが返ってきている。しかしこのような処理となっている為，Query で現在の資産状況を確認しても，実際にリクエストした全ての取引がまだ行われていないことがあり，リアルタイムの現状把握ができない状況が発生している。このバッチ処理で行う方式はリクエストが多くなればなるほど，バッチ処理が間に合わずリアルタイム性が悪くなり，取引リクエストからのラグが長くなる。またこのバッチ処理が走っている時にリクエストが行われると，データストアへの書き込みがロックされるため，データストアに書き込む必要があるタイミングと重なるとレスポンスタイムが特に伸びる。Fabric ではその為ワーカ数が多くなりリクエスト数が多くなるとリクエストの失敗こそないが，リアルタイム性が悪くなり，一定のレスポンスタイムの跳ね上がりに覚悟が必要となる。一方 Cassandra ではリクエスト数が多くなると，リクエストが失敗する数が増えるが，それによって一定のレスポンスタイムが確保されリクエスト数が多くなってもレスポンスタイムは大きく変化しない。

構成するピアの数が多くなったケース 3 でも同様のことが起きている。ピア数が増えても Fabric と Cassandra 共にレスポンスタイムの平均値，中央値がケース 2 のときと大きく変わらない。これは一貫性をとるための通信などはブロードキャストで送信されるためである。一方でピアの数が増えているため，同じワーカ数でもネットワーク全体としての取引の数が増えていることから，ケース 2 で起きた影響が早くでている。Fabric の Batch PBFT のラグつまりリアルタイム性の低下が早い段階で発生し，最終的にラグとして長くでている。Cassandra ではケース 3 のピア数が 8 と多くなりワーカ数を

64 と多くのリクエスト，つまり多くの取引を 60 秒の間で行ってもレスポンスタイムは大きく変化せず，リクエストの失敗数こそ増えるが，平均値，中央値共に，ほぼケース 2 のワーカ数 1 のときとほぼ同じになっている。

5.2 実装

同様のアプリケーションを Fabric と Cassandra それぞれで作成するにあたり，Fabric を利用したアプリケーションで想定されている利用方法に則ったアプリケーションであったため，実装しなくてはいけないプログラムが Cassandra を利用したアプリケーションのほうが多かった。具体的には Fabric ではアプリケーションピアでは配布されている Docker の起動時に構成にあった設定を記述するのと，インターフェイスとなるチェーンコードの実装だけですんだ。このチェーンコードはデータ設計にもなっており，データ設計の手間も削減されている。またチェーンコードの実装も複雑なことはなく，用意されているライブラリを利用すると簡単に実装できた。一方でこのチェーンコードの実装がデータ設計でありデータベースのテーブルの作成のようなものとなるため，ここで定義してある以上，途中で変更ができない。CA ピアの設定も Fabric では設定ファイルに数行書くだけで済んだ。

Cassandra を利用したアプリケーションでは，CA ピアに Fabric で実行されていた代わりとなる認証を行うプログラムと，チェーンコードの代わりとなる API のプログラム，Cassandra の設定，そしてデータの設計が必要であった。実装する必要があるプログラムが増えることは確かだが，Cassandra が扱われている事例なども既にあり，そこまで難しいものではなかった。その一方で柔軟な利用が可能である為に慎重である必要がある。実装によって一貫性レベルの変更が可能でもあり，十分に注意し利用したい条件に合わせた実装が必要である。

5.3 運用

まずネットワークを構成するピアを増やしたり減らしたりする際の差を見ていく。Fabric ではまず全てのピアを知っている必要があり，その数によって設定ファイルにコンセンサスを取る際の設定を記述するため，一度運用し始めるとネットワークを作り直さない限り，ピアの数を増やしたり減らしたりすることが出来ない。一方で Cassandra では一度作成したネットワークにピアを追加したり取り除いたりすることが容易にできる。

またピア一つ一つの運用として，Fabric では全てのデータを全てのピアで持ち，取引毎にデータが増え，データ量の増加限りはないという特性から，容量がいっぱいになってしまったときのことを考慮しなくてはいけない。

5.4 まとめ

評価実験を行うことで Fabric と Cassandra の違いが様々な形で現れる結果となった。特に性能評価ではエラーを返すことで一定のレスポンスタイムを確立しているCassandraと，バッチ処理で後から一貫性を担保し，全てのピアで全てのデータを持ち，リアルタイムではないかもしれないがリクエスト自体は高い確率で素早く返ってくる Fabric と，大きな違いがあった。またブロックチェーンの特性上，どの会社がどれくらい資産を所要しているかという，既存のデータベースではデータ量の変わらない情報を扱っていても，取引を行う毎にデータが増えていくため，トランザクション処理が多く発生し，マシンの容量を圧迫するという問題があった。

このバッチ処理は厄介であり，バッチ処理が行われている途中にもしピアに障害が起きてしまうと，そのピアにリクエストされていた取引は全てなかったことになってしまい，どこまで取引がされたか把握できないという問題も発生する。つまり Fabric の現状において資産を確認するという機能の可用性が低いだけでなく分断耐性も弱いことがわかった。このバッチ処理で行われているコンセンサスモデルは今後選択可能になると公式に告知されており，Cassandra の方式に近いものへとなっていくと考えられる。

現状でいうと Cassandra は既にメジャーバージョンでリリースされており，大きな企業での導入事例もあるが，一方で Fabric はまだメジャーバージョンのリリースはなく開発途中であると大きく異る。この差が性能評価の結果をもたらしている一面はあり，特にバッチ処理の際に発生していることは考慮する必要がある。一方でブロックチェーン自体の仕組みに依存する結果も多く発生しているので結果を無視することは出来ないことも事実である。

以上のことから分散アプリケーションにおける汎用的なミドルウェアの選定としてみた際には，実装には自由度から正確な把握と設定が必要とされるが，運用面と性能の安定面，可用性などを加味するとCassandra の選択が特に現状においては優先されると考えられる。

6 将来展望

性能評価から現状において Fabric は汎用的に利用するにはミドルウェアとしてはまだ未成熟であり，汎用的にブロックチェーンという仕組みを利用しなければいけない正しい利用場面も難しいことがわかった。このことから，汎用的ブロックチェーンの利用には分散データベースという一面よりも，公正な取引を第三者介さずに行え，データを取得する際に他のピアに問い合わせが発生しないことを最大限に活かした利用が重要となる。ブロックチェーンという仕組みではなくとも信頼のある取引の仕組みづくりは可能であり，また全てのデータを全ピアで所有するということはデータの作成や更新つまり今評価実験の NEW，EXECUTE でなくデータの取得今評価実験の QUERY のみが分散されるということであり，魅力としてはいまひとつである。一方で信頼のある取引の仕組みがミドルウェアとして提供されると，最も提供が難しい信頼が担保されるためこれは大きなメリットとなる。

このプロジェクトが多くの企業において作られ，オープンにされていることから Fabric というブロックチェーンは一定の企業が信頼する仕組みとして提供される。これが最も重要な点となっていくと考えられる。また Fabric は分散システムの通信に RPC[17]のフレームワークとしてグーグルが提供する gRPC[8]も採用されている。その為多くの組織で運用していく際に利点となる。

ブロックチェーンは Bitcoin のような不特定多数から成り立つネットワークの時に必要だった機能が多い。そこから許可制のブロックチェーンではブロックチェーンのデータ構造が主に重要となっている。許可制のブロックチェーンにすることでコンセンサスモデルからなる問題は解決を目指されている。一方で今回問題となったデータ量の増加とそれに伴うデータの永続化への処理の増加は物理的マシンによる対応のみが考えられる。Cassandra のようなレプリケーションを取るがデータをある程度分散させる仕組みを取り入れ，また企業の取引に於いては企業が関わるデータなどピア毎に来るリクエストは偏ることが想定できるため，想定の元データの分散を行えば，データの永続化を行うようなトランザクション処理の分散とデータ自体の分散も可能かもしれない。

ブロックチェーンの汎用性とは，小売店など小さな企業でも大企業同様の信頼が担保された仕組みを使えるというものであるように感じる。しかし小売店のような小さな企業でも多くの企業との取引が考えられるため，取引は多くなる。その為大企業に近い多くのデータ処理とデータを扱うことのできるマシンのスペックも必要となる。これは少しハードルが高く感じるが，これはクラウド化が進んでいることから解決できるかもしれない。

ブロックチェーンの他のメリットとして，チェーンとなっている取引の履歴を常に取扱い，データの一貫性は現在の状況だけでなく過去からの変更全てでとれていることがある。現在の状態になった手順も他のピアとコンセンサスを取っており，情報を共有する。この仕組みは利用する人からも大きな機能となり得り，既存の分散データベースとの特に大きな違いとなる可能性をもつ。この機能を汎用的に利用するのは難しいが，有効に利用できる場面がくれば大きなメリットとなる。

[8] http://www.grpc.io/

7　結論

ブロックチェーンに汎用的な利用が期待されている背景から，本研究では汎用的な利用が期待されている許可制ブロックチェーンと既存の分散データベースとを比較することで，正しい利用方法の違いを明確にすることを目指した。そこで今研究では許可制ブロックチェーンとして Fabric を選択し，既存の分散データベースとして Cassandra を選択し，同じ機能を持つアプリケーションの作成を行うことで比較をした。その結果現状の Fabric で実装されているコンセンサス方法である Batch PBFT の特性から，構成するネットワーク全体の取引量がリクエスト数の増加や構成するピアの増加などにより増加すると，リアルタイム性が失われていく代わりにリクエスト自体はすぐに返ってくることがわかった。またブロックチェーンの取引する毎にデータが増えていく特性からトランザクション処理が多く走ることがわかった。一方で Cassandra を用いて作成したアプリケーションでは，ネットワークを構成するピアが多くなり，そのピアへ多くリクエストが行われても，少ないエラーを返すことで安定した性能を保つことができることがわかった。このことから Fabric は Cassandra より早いレスポンスを返してはいることが多いが，安定した動作を求めるのであれば Cassandra を選択することを推奨する。実装面では Fabric の上に乗ってしまえば少ない設定とチェーンコードと呼ばれるプログラムを書くことで済むが，Cassandra では必要に応じて Fabric が提供する多くの機能を実装しなくてはいけない。また Cassandra は一貫性のレベルを設定できるなど，実装方法によって性能が変わってくる為，誰もが安定した利用をできるという観点で Fabric が優れていることがわかった。運用面は Fabric では許可制ブロックチェーンであるために一度作成したネットワークに対してのピアの増減は容易ではなく，また取引が起きる毎にデータが増えていく。そしてデータはピア間で分散することなく全てのピアで所有する。一方で Cassandra では一度作成したネットワークにもピアを簡単に増減でき，一度作成したデータ設計以上にデータは増えずデータの分散もレプリケーションレベルを自由に設定し行うことができることから Cassandra のほうが運用し易いことがわかった。

以上の結果から汎用的なブロックチェーンの利用は現状において必要とされる場面が多くないが，今後プラットフォームとして確立された際には，共通の信頼性のある取引が行えるシステムの開発を安定した品質で簡単にできるようになり，これが最も重要であることがわかった。

8 参考文献

[1] TAO, Fei, et al. Cloud manufacturing: a computing and service-oriented manufacturing model. Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture, 2011, 225.10: 1969-1976.

[2] CISCO, Cisco Visual Networking Index. Global Mobile Data Traffic Forecast Update, 2015-2020 White Paper, 2016.

[3] TKACH, Robert W. Network traffic and system capacity: Scaling for the future. In: Optical Communication (ECOC), 2010 36th European Conference and Exhibition on. IEEE, 2010. p. 1-22.

[4] CHRISTIDIS, Konstantinos; DEVETSIKIOTIS, Michael. Blockchains and Smart Contracts for the Internet of Things. IEEE Access, 2016, 4: 2292-2303.

[5] CACHIN, Christian. Architecture of the Hyperledger blockchain fabric. In: Workshop on Distributed Cryptocurrencies and Consensus Ledgers. 2016.

[6] NAKAMOTO, Satoshi. Bitcoin: A peer-to-peer electronic cash system. 2008.

[7] BENTOV, Iddo, et al. Proof of Activity: Extending Bitcoin's Proof of Work via Proof of Stake [Extended Abstract] y. ACM SIGMETRICS Performance Evaluation Review, 2014, 42.3: 34-37.

[8] CASTRO, Miguel, et al. Practical Byzantine fault tolerance. In: OSDI. 1999. p. 173-186.

[9] GILBERT, Seth; LYNCH, Nancy. Perspectives on the CAP Theorem. Computer, 2012, 45.2: 30-36.

[10] COWLING, James, et al. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In: Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006. p. 177-190.

[11] ÖZSU, M. Tamer; VALDURIEZ, Patrick. Principles of distributed database systems. Springer Science & Business Media, 2011.

[12] CHANG, Fay, et al. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 2008, 26.2: 4.

[13] DECANDIA, Giuseppe, et al. Dynamo: amazon's highly available key-value store. ACM SIGOPS operating systems review, 2007, 41.6: 205-220.

[14] LAKSHMAN, Avinash; MALIK, Prashant. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 2010, 44.2: 35-40.

[15] KARGER, David, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. ACM, 1997. p. 654-663.

[16] PIKE, Rob. The go programming language. Talk given at Google's Tech Talks, 2009.

[17] NELSON, Bruce Jay. Remote procedure call. Carnegie-Mellon Univ. Dept. Comput. Sci., 1981.

9　謝辞

本研究を行うにあたってこのような機会を設けていただき,また的確なご指導を賜りました中島達夫教授に心より感謝いたします。また，本研究のアドバイスや相談に乗っていただいた研究室の先輩方・同輩たちに感謝します。

10 付録

```
membersrvc:
  image: hyperledger/fabric-membersrvc
  ports:
    - "7054:7054"
  command: membersrvc
```

付録 1 Fabric Memberservc 設定ファイル

```
vp1:
  image: hyperledger/fabric-peer
  ports:
    - "7050:7050"
    - "7051:7051"
    - "7052:7052"
    - "7053:7053"
  hostname: vp1
  environment:
    - CORE_PEER_ID=vp1
    - CORE_PEER_ADDRESSAUTODETECT=true
    - CORE_PEER_NETWORKID=dev
    - CORE_VM_ENDPOINT==unix:///var/run/docker.sock
    - CORE_LOGGING_LEVEL=INFO
    - CORE_PEER_PKI_ECA_PADDR=54.249.54.193:7054
    - CORE_PEER_PKI_TCA_PADDR=54.249.54.193:7054
    - CORE_PEER_PKI_TLSCA_PADDR=54.249.54.193:7054
```

```
  - CORE_PEER_DISCOVERY_ROOTNODE=54.202.245.216:7051,54.93.90.83:7051…
  - CORE_PEER_VALIDATOR_CONSENSUS_PLUGIN=pbft
  - CORE_PBFT_GENERAL_MODE=batch
  - CORE_PBFT_GENERAL_N=4
  - CORE_PBFT_GENERAL_K=2
  - CORE_PBFT_GENERAL_BATCHSIZE=2
  - CORE_PBFT_GENERAL_TIMEOUT_REQUEST=2s
  - CORE_PBFT_GENERAL_TIMEOUT_VIEWCHANGE=2s
  - CORE_SECURITY_ENABLED=true
  - CORE_SECURITY_ENROLLID=test_vp1
  - CORE_SECURITY_ENROLLSECRET=5wgHK9qqYaPy
volumes:
  - /var/run/docker.sock:/var/run/docker.sock
command: peer node start --peer-chaincodedev
```

付録 2 Fabric Peer 設定ファイル

```
cluster_name: 'Tom Cluster'
num_tokens: 256
hinted_handoff_enabled: true
max_hint_window_in_ms: 10800000
hinted_handoff_throttle_in_kb: 1024
max_hints_delivery_threads: 2
hints_flush_period_in_ms: 10000
max_hints_file_size_in_mb: 128
batchlog_replay_throttle_in_kb: 1024
authenticator: AllowAllAuthenticator
authorizer: AllowAllAuthorizer
role_manager: CassandraRoleManager
roles_validity_in_ms: 2000
permissions_validity_in_ms: 2000
credentials_validity_in_ms: 2000
partitioner: org.apache.cassandra.dht.Murmur3Partitioner
data_file_directories:
    - /var/lib/cassandra/data
commitlog_directory: /var/lib/cassandra/commitlog
disk_failure_policy: stop
commit_failure_policy: stop
prepared_statements_cache_size_mb:
thrift_prepared_statements_cache_size_mb:
key_cache_size_in_mb:
key_cache_save_period: 14400
row_cache_size_in_mb: 0
row_cache_save_period: 0
counter_cache_size_in_mb:
counter_cache_save_period: 7200
saved_caches_directory: /var/lib/cassandra/saved_caches
commitlog_sync: periodic
commitlog_sync_period_in_ms: 10000
commitlog_segment_size_in_mb: 32
```

```
seed_provider:
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          - seeds: "54.202.229.46"
concurrent_reads: 32
concurrent_writes: 32
concurrent_counter_writes: 32
concurrent_materialized_view_writes: 32
memtable_allocation_type: heap_buffers
index_summary_capacity_in_mb:
index_summary_resize_interval_in_minutes: 60
trickle_fsync: false
trickle_fsync_interval_in_kb: 10240
storage_port: 7000
ssl_storage_port: 7001
listen_address: 172.31.36.117
broadcast_address: 54.202.229.46
start_native_transport: true
native_transport_port: 9042
start_rpc: true
rpc_address: 0.0.0.0
rpc_port: 9160
broadcast_rpc_address: 54.202.229.46
rpc_keepalive: true
rpc_server_type: sync
thrift_framed_transport_size_in_mb: 15
incremental_backups: false
snapshot_before_compaction: false
auto_snapshot: true
column_index_size_in_kb: 64
column_index_cache_size_in_kb: 2
compaction_throughput_mb_per_sec: 16
```

```
sstable_preemptive_open_interval_in_mb: 50
read_request_timeout_in_ms: 5000
range_request_timeout_in_ms: 10000
write_request_timeout_in_ms: 2000
counter_write_request_timeout_in_ms: 5000
cas_contention_timeout_in_ms: 1000
truncate_request_timeout_in_ms: 60000
request_timeout_in_ms: 10000
cross_node_timeout: false
endpoint_snitch: SimpleSnitch
dynamic_snitch_update_interval_in_ms: 100
dynamic_snitch_reset_interval_in_ms: 600000
dynamic_snitch_badness_threshold: 0.1
request_scheduler: org.apache.cassandra.scheduler.NoScheduler
server_encryption_options:
   internode_encryption: none
   keystore: conf/.keystore
   keystore_password: cassandra
   truststore: conf/.truststore
   truststore_password: cassandra
client_encryption_options:
   enabled: false
   optional: false
   keystore: conf/.keystore
   keystore_password: cassandra
internode_compression: dc
inter_dc_tcp_nodelay: false
tracetype_query_ttl: 86400
tracetype_repair_ttl: 604800
enable_user_defined_functions: false
enable_scripted_user_defined_functions: false
windows_timer_interval: 1
```

```
transparent_data_encryption_options:
    enabled: false
    chunk_length_kb: 64
    cipher: AES/CBC/PKCS5Padding
    key_alias: testing:1
    key_provider:
      - class_name: org.apache.cassandra.security.JKSKeyProvider
        parameters:
          - keystore: conf/.keystore
            keystore_password: cassandra
            store_type: JCEKS
            key_password: cassandra
tombstone_warn_threshold: 1000
tombstone_failure_threshold: 100000
batch_size_warn_threshold_in_kb: 5
batch_size_fail_threshold_in_kb: 50
unlogged_batch_across_partitions_warn_threshold: 10
compaction_large_partition_warning_threshold_mb: 100
gc_warn_threshold_in_ms: 1000
```

付録 3 設定ファイル cassandra.yaml